University of Alberta

Reverse Engineering Heterogeneous Software Systems

by

Daniel Leontin Moise

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**

Department of Computing Science

Edmonton, Alberta, Canada
Spring 2007

Library and Archives Canada

Published Heritage Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et Archives Canada

Direction du Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 978-0-494-29997-5
Our file Notre référence
ISBN: 978-0-494-29997-5

NOTICE:
The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:
L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canada

To my lovely wife, Gabriela, who constantly encouraged me.

Abstract

Nowadays, the abundance of new technologies and languages used to ease application development raises new challenges for reverse engineers. During development, programmers need to understand not only the dependencies among code in a particular language, but dependencies that span languages. This thesis studies the problem of finding, representing and visualizing cross-language dependencies in such diverse, heterogeneous software systems. The goal of this thesis is to develop techniques for helping engineers to understand and navigate multi-language software systems.

Our approach involves building a fact extractor for each language. The produced facts conform to a common schema, and an analyzer is extended to recognize the cross-language dependencies. We present how these statically discovered dependencies can be represented, visualized, and explored in our developed tool called Clare, which is a plugin in Eclipse. Some tests illustrate the usefulness and scalability of our approach.

Acknowledgements

I would like to thank my supervisor, Dr. Kenny Wong, for his help and support. He gave me a chance, and he encouraged me in all this time. Thanks a lot Ken!

I would like to thank the members of my committee, Dr. James Hoover and Dr. Marek Reformat, for their insightful feedback.

Also, I would like to thank Dr. Daqing Hou for our long discussions and the help that he gave me.

Contents

1 Introduction 1
- 1.1 Motivation . . . 2
- 1.2 Related Work . . . 3
- 1.3 Contribution . . . 4
- 1.4 Outline . . . 5

2 Background 6
- 2.1 Reverse Engineering . . . 6
- 2.2 Summary . . . 9

3 Overall Approach 11
- 3.1 Technique . . . 11
- 3.2 Architecture Overview . . . 13

4 Extracting Facts for C/C++ and Java 15
- 4.1 Source Navigator . . . 16
- 4.2 Output Formats . . . 19
  - 4.2.1 Graph eXchange Language (GXL) Representation . . . 19
  - 4.2.2 Clare XML-based Representation . . . 21
- 4.3 Extracting the Facts . . . 21
  - 4.3.1 Populating Source Navigator Internal Database . . . 22
  - 4.3.2 Producing the Output Factbase . . . 24
- 4.4 C/C++ and Java Schemas . . . 26
- 4.5 C/C++ and Java Results . . . 30
  - 4.5.1 C/C++ Factbase Example . . . 31
  - 4.5.2 Java Factbase Example . . . 31
- 4.6 Summary . . . 31

5 Extracting Facts for Perl 34
- 5.1 Perl Internals . . . 35
  - 5.1.1 Perl Data Types . . . 37
  - 5.1.2 Perl Subroutines . . . 40
  - 5.1.3 Perl Namespaces . . . 43
- 5.2 Perl Extractor . . . 44
  - 5.2.1 Interpreting Perl Scripts . . . 44

5.2.2 Perl Interpreter Control Flow 46
5.2.3 Perl Extractor Implementation 47
5.3 Perl Schema 49
5.4 Perl Results 50
5.4.1 Perl Factbase Example 50
5.4.2 Perl Extractor Test 51
5.5 Summary 53

6 Analyzing and Representing Cross-Language Dependencies 54
6.1 Recognizing Java to/from C/C++ 54
6.2 Connecting Perl to C 59
6.3 Connecting Tcl to C 62
6.4 Connecting Python to C 64
6.5 Extension Mechanism Commonalities 66
6.6 Common Schema 67
6.7 Summary 69

7 Evaluation and Applications 71
7.1 Perl B Module Test 71
7.2 Win32RegKey Test 72
7.3 Java GNome Test 76
7.4 Standard Perl Modules Test 77
7.5 Summary 78

8 Related Work 79

9 Conclusions and Future Work 86

Bibliography 88

List of Figures

2.1 The place of reverse engineering in the phases of life cycle of a software development process [Chikofsky90] 7

3.1 Overall Architecture 13

4.1 Source Navigator Architecture 16
4.2 Source Navigator Database Access 17
4.3 GXL representation example 20
4.4 Clare representation example 21
4.5 Snippet from the extractor implementation 25
4.6 A simple C++ example 31
4.7 Factbase for the C++ example in Clare representation 32
4.8 A simple Java example 32
4.9 Factbase for the Java example in Clare representation 33

5.1 Internal Perl data type hierarchy 38
5.2 General Perl *SV* structure 38
5.3 Perl operation structure hierarchy 41
5.4 A simple Perl script 42
5.5 Perl operation tree 42
5.6 Perl internal stashes representation 44
5.7 Perl interpreter architecture 45
5.8 Overview of Perl call tree 46
5.9 Perl extractor pseudocode 48
5.10 Perl schema 50
5.11 Factbase for the Perl example in Clare representation 52

6.1 *Test* Java class 55
6.2 Algorithm for finding cross-dependencies from Java to C/C++ 57
6.3 C sample code for accessing Java from C code 58
6.4 Algorithm for finding cross-dependencies from C/C++ to Java 59
6.5 Listing of *Test.c* file for Perl 60
6.6 Listing of *Test.pm* Perl module file 62
6.7 Listing of *Test.c* file for Tcl 63
6.8 Listing of *Test.c* file for Python 65

7.1 Support Perl-to-C dependencies in Eclipse 72

7.2 *Win32RegKey* source code snippet 74
7.3 Snippets of *Win32RegKey* Java factbase 75
7.4 Snippets of *Win32RegKey* C factbase 75
7.5 Snippets of *Win32RegKey* result cross-language dependencies factbase 76
7.6 Java-GNome mistakes 78

List of Tables

4.1 Source Navigator types used for predefined languages 18
4.2 Source Navigator internal database format 23
4.3 C/C++ Schema - Node and Relation Types 28
4.4 C/C++ Schema - Attribute Types 29
4.5 Java Schema 30
4.6 Java Schema - Attribute Types 30

5.1 Perl data type correspondence 37
5.2 Selected Perl operations 41

6.1 Mapping Java and C/C++ Types 56
6.2 Encoding Java Signatures 58
6.3 Summary of Perl, Tcl and Python to C extension mechanisms 66
6.4 Common schema entity types 67
6.5 Common schema relation types 68
6.6 Common Schema - Attribute Types 69

Chapter 1

Introduction

Chikovsky et al. [Chikofsky90] defined reverse engineering as being the "the process of analyzing a subject system in order to identify the system's components and their interrelationships, and to create representations of the system, possibly at a higher level of abstraction."

The input of a reverse engineering process might be anything regarding the subject software system, such as: source code, building files, design diagrams, specifications, user manuals, log files. This process uses tools and/or techniques to ease the work of software developers, engineers or maintainers solving a specific task. The goal of this process is to provide different perspectives of the software system based on the implementation for a better comprehension of the system. This should have a positive impact for the process of maintenance and evolution of the product.

Most of the reverse engineering tools focus on finding the dependencies within a system written in a single programming. Nowadays, with the advent of new technologies, many systems are implemented using more than one language. Therefore, the developers need urgently tools and techniques to understand these heterogeneous applications.

In this thesis we present an approach for finding, representing and visualizing dependencies in software systems written in more than one language.

As a simple example, let's consider a multi-language application that is coded using both Java and C languages. Most of the existing tools provide the code dependencies only within each language, Java or C, but not across the languages, Java to C or C to Java. Our approach finds, besides the *intra*-language dependencies, the *inter*-language dependencies where Java code is accessing C code, or C code is accessing Java code, for example.

From now on, we refer to a software system written in more than one language using one of the following terms: multi-language, mixed language, diverse, or heterogeneous. We also use the term cross-dependencies or inter-dependencies for the connections among different languages. Cross-dependencies do not include the connections inside the same language (referred as intra-dependencies).

1.1 Motivation

The diversity of programming languages, technologies, and platforms used in modern software systems today creates significant understanding, evolution, and management challenges. Software developers have moved far beyond support for a single language or platform, and are in urgent need of tools that can analyze multi-language, multi-platform systems [Baxter, Lammel01, Linos03, Eichberg05]. For instance, addressing this need is critical for the long-term viability of Web-based applications [Hassan01], which may contain a mix of code in Java, HTML, JavaScript, VBScript, SQL, etc. In addition, there is a vast amount of heterogenous legacy software that must still be maintained and/or migrated to modern environments. Many systems are written with entity, control, and boundary layers, each implemented or generated by a different suitable language. Ubiquitous .NET applications allow developers to access the .NET framework by writing code in different languages.

The goal of most programming languages is to be "the one language needed to meet all of your programming needs", but the reality is that no one language is sufficient to satisfy all the developer's needs. For example, developers build systems from components written in different programming languages, by gluing them together into a single system. In this case, all components share information about the interfaces of the objects, typically specified in another language called IDL (Interface Description Language) [IDL]. Scripting languages are another example, since they provide different techniques to be extended by/embedded in software written in another programming language.

There are a number of reasons why multi-language systems exist.

- **Efficiency**

  For performance reasons, a high-level language may invoke fragments of code in another lower-level language (e.g., C with embedded assembly). An interpreted language may call functions written in a natively compiled language (e.g., Perl with calls to a C library).

- **Suitability**

  For certain tasks, some languages and notations may be more suitable than others. For example, SQL (Structured Query Language) is the standard notation for manipulating relational data. Scripting languages are useful in gluing together programs. In particular, Perl is very effective at text processing.

- **Reuse**

  A software system may need to interoperate with another one as is, even if written in another language, rather than rewriting everything into a single language. The different teams working on each system may continue to use the language with which they are most familiar.

For a diverse system, analyzing the software written in each language as an island unto itself is not sufficient. These analyses need to be connected together or bridged to create a comprehensive, more integrated picture [Deruelle01, Kullbach98]. For example, programmers often need to follow control flows in software, and this activity should not be constrained by language boundaries. It would be useful to know if, say, a C function was ultimately called from Perl code to better assess the impact of potential changes. Also, a more integrated understanding can help in looking for inconsistencies or anomalies, such as malformed or missing stubs in the cross-language mechanism. If a C function is declared to be called from Perl, a static program analysis can check that the C function indeed exists. Moreover, the integrated analyses need to be presented in a seamless form that is familiar to developers, such as in the context of the Integrated Development Environments (IDEs) (Eclipse [Eclipse], Visual Studio .NET, and so on). Finally, a comprehensive understanding can aid in recovering the system architecture [Hassan02a, Hassan02b].

1.2 Related Work

This section provides a short overview of the existing research work on tools for analyzing multi-language software systems. See Chapter 2 for more details.

Linos et al. [Linos03] implemented a prototype tool called MT (Multi-Language Tool) for understanding multi-language program dependencies. The purpose of MT is to ease the process of detecting, storing, and managing MLDPs (Multi-Language Program Dependencies) found in programs written using a combination of C, C++, and Java languages. The extractor used in this tool is based on a lexical analysis. We believe that a syntactic extractor with more precise facts is needed for analyzing such heterogeneous systems.

Hassan et al. [Hassan01, Hassan02a, Hassan02b, Hassan03] proposed a methodology for maintaining Web applications. A set of extractors is used to analyze the source code of Web applications. The outcome of this analysis is a set of relationships between various components of a Web application. This work focuses more on extracting the architecture of a web system.

Deruelle et al. [Deruelle01] described a method for analyzing distributed multi-language software systems. Several tools help to accomplish this: a multi-language source code analyzer, a software change management module, a profiling tool, and a graphical user interface. The multi-language source code analyzer consists of a set of parsers for each of the languages considered (C, C++, and Java). Each parser is generated using the JavaCC tool based on a language-specific grammar. The input could also be provided as bytecode, in which case a decompiler is run first. The dependencies between languages are extracted by parsing IDL files. This approach focuses more on analyzing CORBA [CORBA] distributed systems.

Kullbach et al. [Kullbach98] described a tool that helps the management of inter-program dependencies for a software application developed in vari-

ous programming languages, database definitions and job control languages. Their approach uses a coarse-grained conceptual model for the individual programming languages, on which an integrated model for the multi-language application is developed. The key observation here is that the inter-program dependencies are defined by job control procedures that coordinate a number of programs and databases. Also, this work focuses more on providing a query language for analyzing multi-language systems.

1.3 Contribution

In this thesis, we present a method to comprehend how diverse software is structured and integrated. There are many mechanisms by which software written in one language can transfer control and data to a separate piece of software potentially written in another language. Our approach recognizes and presents such cross-language dependencies.

Our approach consists of several tools for analyzing multi-language software. A set of extractors is used to extract the facts for each individual language present in the application based on the source code. The factbases conform to a common schema. Essentially, a multi-language system can be represented as a set of namespaces, with each containing facts from one language. The common schema helps to decouple the cross-language dependency analysis (and downstream tools like visualizers) from the individual language fact extractors. The factbases feed different recognizers of cross-language dependencies. A visualization tool was developed to present the relations inside each factbase as well as the cross-language dependencies. The recognizers of cross-language dependencies and the visualization are under a tool, called Clare. Clare was developed as a plug-in in Eclipse [Eclipse] to alleviate the problem of adoption for the tool. Therefore, the developers can use Clare to understand the cross-language dependencies at the same time while developing heterogeneous applications.

In summary, the main contributions of this thesis are:

- extend extractors for C, C++, Java;
- build an extractor for the Perl scripting language;
- introduce a common schema for code artifacts across several languages;
- provide different recognizers of cross-language integration or linkage mechanisms between pairs of languages;
- develop a visualization tool to view and explore the cross-language dependencies under the Eclipse platform;
- reveal some mistakes made by developers when developing a software system written in Java and C.

Our toolset helps developers to understand and maintain better heterogeneous software system, by providing the automatically extracted cross-dependencies between the languages, and by allowing the exploration of calls not only within each language, but also across the languages.

The results of our research are published in several top conferences in reverse engineering [Moise03, Moise04a, Moise04b, Moise05, Moise06a, Moise06b].

1.4 Outline

The rest of the thesis is organized as follows. Chapter 2 introduces the reverse engineering in the life of a software system. Chapter 3 presents the technique used and the high-level architecture of our approach. In Chapter 4 and Chapter 5, we describe the extractors used for individual languages. Chapter 6 describes the recognizers for several mechanisms used to transfer the control and data between different languages, and presents a common schema used to represent the facts. The evaluation of our approach on some case studies is presented in Chapter 7. Chapter 8 presents some related work for different types of extractors. Finally, Chapter 9 proposes directions for future research and draws the conclusions.

Chapter 2

Background

This chapter presents the background and the related work with regard to this thesis. We present the place and need of the reverse engineering process in the software development life-cycle in Section 2.1.

2.1 Reverse Engineering

Reverse engineering has its origin in the analysis of a hardware system, with the aim of duplicating the original hardware system. Well known stories travel around the world about "stealing" technology from an enemy or from a competitor company using reverse engineering of hardware systems. Probably the most known story of reverse engineering a hardware system is how the soviets copied the American's best bomber during the Second World War [CNN]. Around 1944, three B-29 American's planes that returned from bombing missions against Japan, made emergency landings in Russia (who was America's ally at that time). The pilots did not know what was going to happen to their planes. The Soviet leader ordered disassembling the planes and to build similar planes for the Soviet army. The planes were copied exactly after the American's ones, even with the problems the planes had. This story remains as the most well known story of reverse engineering a hardware system.

The same term of reverse engineering used in software engineering has the meaning of understanding the design of a software system in order to improve the maintenance, enhancement or even replacement of the software product. While a company reverse engineers hardware systems to understand enemy or competitor products, a company may reverse engineers software systems to understand its own work. The reason is that in many cases the documentation and specifications for the software product do not exist. From now on, we use the reverse engineering term to refer to reverse engineering software systems.

Chikofsky and Cross present a taxonomy of terms (forward engineering, reverse engineering, redocumentation, design recovery, restructuring and reengineering) with respect to the life-cycle phases and activities in which they are involved [Chikofsky90]. Three fundamental activities exist in the life cycle of

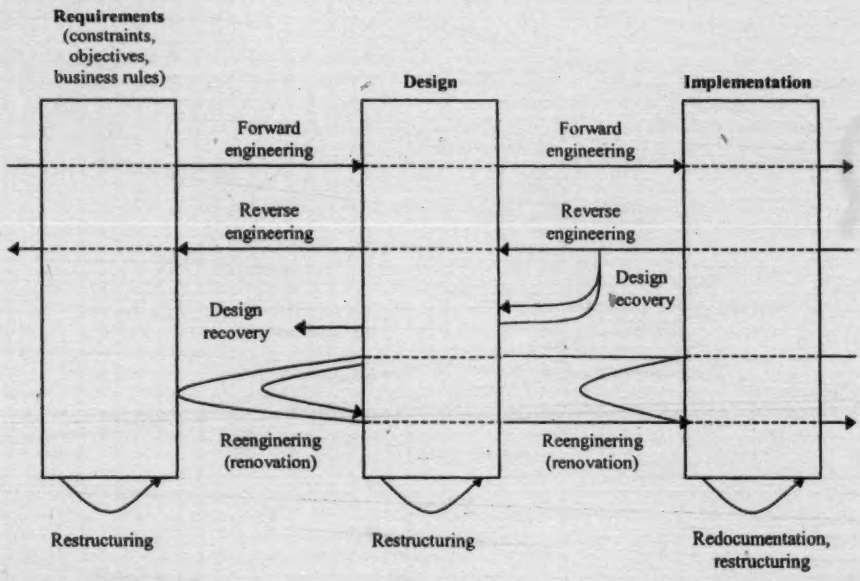

Figure 2.1: The place of reverse engineering in the phases of life cycle of a software development process [Chikofsky90]

every software development process:

- *requirements analysis*: what the product should do and under which constraints
- *design*: how the system will be implemented - describing a model for the established requirements
- *implementation*: coding, testing, debugging and providing the solution.

Figure 2.1 illustrates the relationships among the processes defined next in this section and the life cycle of a software development cycle.

Forward engineering is the process of moving from the high-level abstractions (from the requirements and design phases) to the low level implementation of the system. This process follows different scenarios depending on the chosen software process used for developing the system. For example, in the traditional process, the sequence of steps is to gather the requirements, then move to analyze the requirements for building the design of the system, and finally to implement the product based on the design. Reverse engineering is the opposite process of forward engineering; thus, reverse engineering process starts from source code (implementation) to recover the design and eventually to provide the initial requirements. The role of reverse engineering is to understand the software system using different specific tools and not to modify or replicate the system. Chikofsky and Cross give the following definition for reverse engineering [Chikofsky90].

> *Reverse engineering* is the process of analyzing a software system in order to identify the system's components and their interrelationships and to create representations of the system, possible at a higher level of abstraction.

Note that reverse engineering is a process of examination only - the software system under consideration is not modified.

Two important subareas of reverse engineering are redocumentation and design recovery, which are defined with respect to the levels of understanding that they achieve. *Redocumentation* is the process of extracting different representations of a system in order to identify certain characteristics existing in different versions. The prefix *re* from redocumentation is coming from the fact that this process extracts the documentation even though other documentation may already exist for the system. The purpose of the redocumentation process is to recover artifacts of the early stages of development; therefore it is not required to obtain any function or purpose of the system. The process that complements the redocumentation process is called *design recovery*. Besides the artifacts from the source code, design recovery considers domain knowledge, external information or other existing or deduced observations that could help in obtaining "meaning" for the system or system's components. The design recovery process could lead to the recovery of the steps from the design phase, and sometimes even the rationality that stands behind the decisions taken.

Two topics related to reverse engineering are restructuring and reengineering. These topics are not subareas of reverse engineering because they are not only examining the code, but are modifying it. *Restructuring* is the process of transforming the source code from one representation to another without changing the functionality of the system. A simple example is moving a software system from a procedural implementation to an object-oriented implementation. This transformation will change massive parts of the code, but it will maintain the behavior of the system. *Reengineering* is a form of restructuring that might modify the functionality of the system. Usually, the reengineering process will follow a reverse engineering process, followed by a forward engineering process for adding new features to the system.

The main goal of reverse engineering a software system is to increase the comprehension of the software system. Some of the objectives of the program understanding are: to cope with complexity, to generate alternate views, to recover lost information, to detect side effects and to synthesize higher abstractions. The need for a better comprehension of the system is required by tasks such as maintenance, reuse or overall quality assurance. *Software maintenance* is the process of "modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a changed environment" [IEEE83]. Therefore, the maintenance process might be categorized in four denominations: corrective (repair the bugs in a system), adaptive (moving the system to a new environment), perfective (adding

new functionality to the system), and preventive (modifications of the system to ease the future changes). In addition to reverse engineering tools, some of these maintenance tasks imply using restructuring or reengineering tools. Reverse engineering tools help to reuse components extracted from the code with the goal of minimizing the cost of developing applications. Also, the reverse engineering tools could help in improving the software quality assurance by providing formal technical reviews, different metrics or validation criteria throughout each phase of the life cycle.

There are studies that reflect the importance of the maintenance process played in software industry:

> "Corporate computer programmers, in fact, now spend 80 percent of their time just repairing the software and updating it to keep it running. Developing new applications in this patchwork quilt has become so muddled that many companies cant figure out where all the money is going." [Carroll88]

> "Estimates that $30 billion is spent each year on maintenance ($10 billion in the United States) with 50 percent of the data processing budgets of most companies going to maintenance and that 50–80 percent of the time of an estimated one million programmers or programming managers is spent on maintenance." [Corbi89] [Parikh87]

> "A Massachusetts Institute of Technology study which indicates that for every $1 allocated for a new development project, $9 will be spent on maintenance for the life cycle of the project." [Corbi89] [Parikh87]

> "... programmers spent an average of 35% of their time simply navigating between dependencies, and an average of 46% of their time inspecting task-irrelevant code." [Ko05]

Keeping in mind that reverse engineering tools help to understand a software system and that "at least half of the maintenance process is understanding the system itself" [Parikh83], we can deduce the necessity of the reverse engineering tools mainly to decrease the costs of the maintenance process.

2.2 Summary

It has been estimated that at least one third of the software applications are written using two different programming languages and that 10% of all software applications use three or more different programming languages [Jones98]. This estimation was made in 1998, so that today it is quite likely that the percentage of the applications using multiple programming languages to be significantly higher than the old percentage. Therefore, program comprehension

and maintenance have become crucial issues when dealing with multi-language software systems.

Nowadays, the abundance of new technologies used to ease the development of medium or large applications raises new challenges for understanding and maintaining these systems implemented in various programming languages. Examples of multi-language software systems are web-based applications, or the legacy systems in which the developers have been using scripting languages to overcome the problem of portability to different platforms. These days, software systems are becoming more and more a "spaghetti" of languages, using low level languages such as C, C++ or Java, to high level ones such as scripting languages (Tcl, Perl, JavaScript). Therefore, the software developers need tools to understand these multi-language systems. However, most of the research in reverse engineering is developed for the legacy systems implemented in only one programming language [Muller93, Wong95, Bowman99, Riva00, Moise03, Antoniol03].

Chapter 3

Overall Approach

Software systems are often written in more than one programming language. During development, programmers need to understand not only the dependencies among code in a particular language, but dependencies that span languages. This chapter presents our approach used to extract and to visualize the cross-language dependencies for diverse, heterogeneous software systems. A high-level overview of the approach is discussed in Section 3.1. A short introduction about the main constituents of the architecture of the approach is given in Section 3.2. Each of the main components will be dissected into more details in the subsequent chapters.

3.1 Technique

Reverse engineering the source code of a software system requires a fact extractor that can identify the constituent entities and relationships. Depending on the nature of the software system, and on the programming languages used, building fact extractors is a difficult task because of the potential need to handle multiple dialects, missing source code, missing libraries, source code for multiple platforms, syntax errors, and so forth. For example, we could have individual extractors for programming languages (such as C, C++ or Java), scripting languages (such as Perl, Tcl or Python), and markup languages (such as HTML, ASP or JSP). A combination of two or more programming languages may exist in the software system. For example, a large software system may embed a scripting language interpreter to ease interoperability with external systems. Also, a Java application can call C functions to access the native platform, to use existing non-Java libraries, or to improve the speed of critical parts of the code. Therefore, for multi-language systems, we need to build extractors that produce dependencies both within and among the constituent languages of the system.

As part of the reverse engineering process, a visualization tool is needed to illustrate the cross-language dependencies. It will be useful to have the visualization tool embedded into a developer environment such as Visual Studio

IDE or Eclipse. Therefore, developers can easily visualize, while implementing diverse software systems, the relationships inside code in each language, as well as the cross language relationships among the code.

The space of languages and cross-language interoperability mechanisms is huge. Rather than considering analyses between every pair of languages, it is helpful to divide the space, narrow our focus, and look for general approaches for each partition.

Consequently, interactions between programmatic entities can be broadly categorized as being either loosely coupled or tightly coupled. Loosely coupled interactions may be enabled by sharing a database or file, communicating through network channels, or invoking procedures remotely through the use of middleware. Such interactions typically cross process boundaries. In contrast, tightly coupled interactions happen within a single process, including both transfers of control and exchanges of data.

Tightly coupled cross-language components may interoperate by providing an interface that is invoked using some common calling mechanism (e.g., C convention with arguments pushed onto the stack in reverse order). Similarly, cross-language components may interoperate if each is compiled into a common intermediate language running on a virtual machine interpreter (e.g., Microsoft Common Language Runtime [CLR], and the Perl 6 Parrot Interpreter [Parrot]). Our approach focuses on studying tightly-coupled cross-language dependencies.

We presents an approach for finding and visualizing tightly-coupled dependencies in multi-language software systems. The approach has three major steps:

1. extract facts from code written in each individual language in the subject diverse software system
2. find the cross-language dependencies among different languages based on the facts discovered in the previous step
3. visualize and explore the cross-language dependencies

The first step of our approach requires building extractors for each individual language from the subject system. We developed extractors for the following languages: C/C++, Java, and Perl. We chose C/C++ and Java for programming languages to show the tightly-coupled connection among programming languages. We chose Perl language to illustrate how to build an extractor for scripting languages, and to show the dependencies between a programming language, such as C, and a scripting language, such as Perl. The first step produces one factbase for each individual language in the diverse system analyzed.

In the second step, we developed an analyzer that recognizes the cross-language dependencies among the factbases obtained in the first step. This step assumes that the factbases obtained in the first step conform to a common

schema. Essentially, a multi-language system can be represented as a set of namespaces, with each containing facts from one language. The schema helps to decouple the cross-language dependency analysis (and downstream tools like visualizers) from the individual language fact extractors.

To explore and visualize the cross-language dependencies, we developed the Clare tool. Clare is implemented as a plug-in for Eclipse IDE, and shows the facts from each individual language, as well as the cross-language dependencies among the languages. Clare can be used to navigate through the call tree of a diverse system without having the limitation of stopping at the border between the languages.

3.2 Architecture Overview

Figure 3.1 illustrates the overall architecture of our approach. Our architecture adopts the standard reference architecture for reverse engineering tools [Ferenc02, Hassan01]. It consists of fact extraction, which contains several extractors for revealing the facts in each language; fact analysis, which embeds several analyzers for finding cross-language dependencies; and fact visualization identified by Clare tool for visualizing, representing, and exploring the facts.

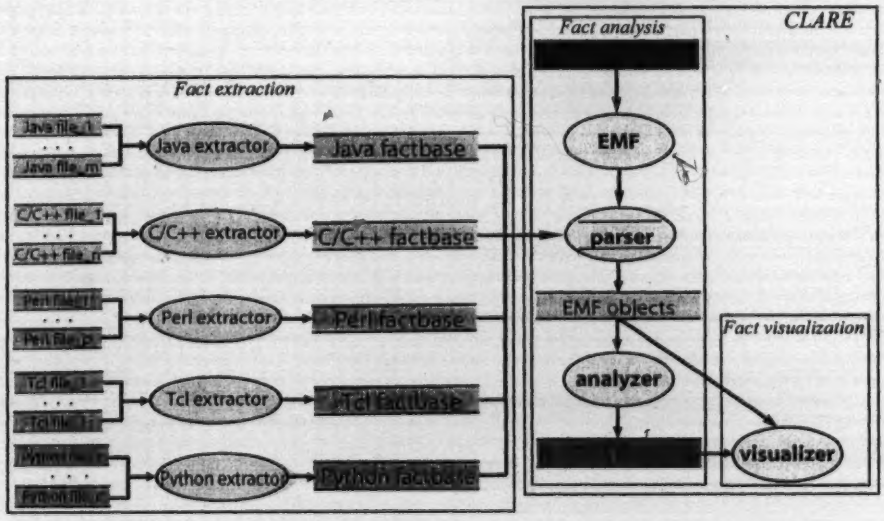

Figure 3.1: Overall Architecture

Given a multi-language software system, an independent factbase is produced for each language involved. Figure 3.1 contains the input files for five different languages: m files for Java, n files for C/C++, p files for Perl, t files for Tcl, and r files for Python. A Perl extractor is used to produce the Perl factbase containing the facts for the input Perl source files. In the same way, a

C/C++ extractor produces the factbase corresponding to C/C++ source files, a Java extractor produces the factbase corresponding to Java source files, and so on.

We developed extractors for three languages using two different approaches. First approach is to use the Source Navigator [SN] front-end, to obtain the facts in a common format understandable by the reverse engineering tools. Source Navigator contains several extractor for different languages, such as C/C++ and Java. The implementation uses predefined extractors in Source Navigator to produce the facts inside each language. Source Navigator extractors are robust, so they work on a variety of software systems without crashing, even when the subject application has syntax errors, or the source code is incomplete. The second approach is to develop a fact extractor for a scripting language by hooking into the language interpreter itself. We developed a Perl extractor using this approach. We believe that the same extraction method can be applied to other scripting languages, such as Tcl. The two approaches used to develop fact extractors is discussed in more detail in Chapter 4, and respectively Chapter 5.

Each factbase produced by one of the extractors is represented in XML format conforming to a common XML schema, and all factbases can be processed by a common parser. A parser reads each factbase associated with a language, and produces an in-memory set of objects that is analyzed to discover cross-language dependencies in the facts. Then, an analyzer identifies the dependencies among different factbases. Chapter 6 describes how the analyzer can find the dependencies between several languages such as C/C++ and Java, or Perl and C.

The facts and dependencies of the studied software system are represented, visualized and explored using Clare tool. The environment of manipulating the facts is Eclipse IDE, which gives the developers the power to develop and explore the software system at the same time, and under the same environment. Clare consists of two main components: one for extracting the cross-language dependencies, and another component that allows to explore and visualize the (cross-)dependencies.

To build Clare, we used the Eclipse platform. In particular, the Eclipse Modeling Framework (EMF) provides services to create and edit data models, as well as a facility to generate code to parse and validate data according to a given schema description. If the data is valid, an in-memory Java object model is constructed automatically. We use EMF to generate the common parser for the XML factbases, using the common XML schema. This approach is especially useful for iterative development. Every time the schema evolves, a new parser can be generated within seconds. Also, the Graphical Editing Framework (GEF) from Eclipse is used to create a rich, graphical editor to present the object model. GEF contains two Eclipse plug-ins: one for graphical drawing and the other for defining an Eclipse workbench window.

Chapter 4

Extracting Facts for C/C++ and Java

Two important characteristics of extractors are their robustness and their accuracy. The robustness of an extractor refers to its ability to deal with different special situations that might appear in the source code. For example, the source code might contain syntax errors or might be incomplete. Sometimes, it is useful to have an extractor that can deal with such situations. On the other hand, an extractor should be as accurate as possible; otherwise the analysis of the software system could lead to mistakes. In practice, when an extractor is built, there is a tradeoff to choose either an extractor that is more robust than accurate, or an extractor that is more accurate than robust. The decision is taken with respect to the reverse engineering tasks that have to be accomplished. Some benchmarks for testing the extractors in terms of accuracy and robustness have been proposed. CppETS (C++ Extractor Test Suite) [Sim02] is such a benchmark that contains a bunch of tests for evaluating the C++ fact extractors.

Another interesting characteristic of an extractor is the format used to store the output file that contains the artifacts of the software system that the extractor produced. Most of the data formats existing in the reverse engineering domain are associated with the reverse engineering tool in which they are used. For example, Rigi tool [Rigi] uses RSF (Rigi Standard Format) [RigiManual], CPPX (C++ Fact Extraction Tool) tool [CPPX] uses TA (Tuple Attribute format) [TA] or GXL (Graph eXchange Language) [GXL], ShriMP (Simple Hierarchical Multi-Perspective) [SHRIMP] uses GXL, RSF, XML [XML] or XMI (XML Metadata Interchange) [XMI]. At the Dagstuhl Seminar on "Interoperability on Reverse Engineering Tools", GXL was ratified as the standard exchange format for the tools in this domain [Winter02], so that nowadays converters have been developed for transforming different formats to GXL.

This chapter presents an approach for generating factbases for different languages using Source Navigator [SN] as a front-end. Source Navigator has

predefined parsers for C, C++, Java, Tcl, [incr Tcl], FORTRAN, and COBOL, and offers the API necessary to add your own parser to this environment.

Our approach uses Source Navigator predefined extractors to produce source code information in a proprietary database format. The Source Navigator API is used to interrogate the Source Navigator database and to produce the factbases in a format accepted by most of the reverse engineering tools. Our extractors output the factbases in two formats: GXL format and an XML-based format supported by the Clare tool. Therefore, this approach ensures the widespread use of the results in the existing reverse engineering tools.

4.1 Source Navigator

Source Navigator is a source code analysis tool that can edit the source code, display the relationships between classes, and functions and members, display call trees, or display artifacts from source files such as functions, variables, "include" files, and so on. Source Navigator could be used, for example, to analyze how a change affects external source modules, to find every place in code where a given function is called, to find each file that includes a given header file, or to search for a given string in all the source files.

There are two major software components in Source Navigator. One component contains the database engine and predefined parsers that fill the database with the facts of a software system. The other component is a graphical user interface (the largest part, i.e., 90%, implemented in Tcl/Tk) for viewing software projects stored in the database. Our approach uses only the first software component to extract the facts from a subject software system in the Source Navigator database. The steps involved in this component are illustrated in Figure 4.1. First, the Source Navigator takes the files associated with a project. Then, based on the filename extensions of the files the appropriate parsers are invoked to extract the artifacts from the source code. Finally, the information extracted in the previous step is stored in an internal database. This process can be executed in two ways: using the Source Navigator interface or using the command line tools provided with Source Navigator.

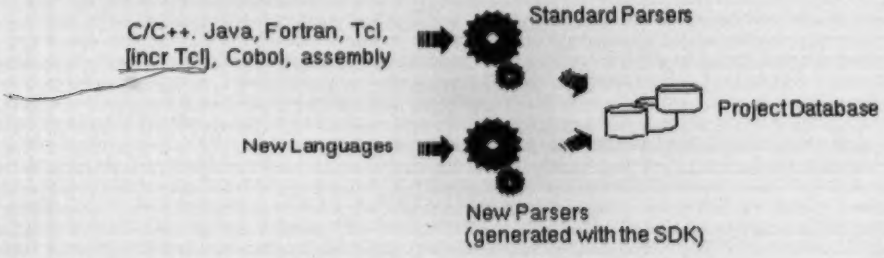

Figure 4.1: Source Navigator Architecture

Source Navigator provides a C API that enables parsers to insert information into a project database. This feature allows writing new parsers for supporting additional programming languages, as illustrated in Figure 4.1. Also, a database API for manipulating the database files and records is provided for both C and Tcl languages. Our approach uses the Tcl API to extract information from the Source Navigator database.

All information associated with a Source Navigator project is stored in an internal database. The Source Navigator database consists of multiple files (between 15 and 25 files for a project), each one representing a table that contains symbol and index information. These files store the information in a binary format using either a sorted and balanced tree structure or an extensible and dynamic hashing scheme. Our approach uses the Tcl API to access the database. The Tcl routines to access the database consists of the following commands:

- *dbopen* - to open a table for reading/writing
- *close* - to close a table de-allocating any resource associated with that table and flushing the cache information to disk
- *del* - to remove key/data pairs from a table using input patterns
- *get* - to fetch information based on index from a table
- *put* - to store key/data in a table
- *isempty* - to check if a table is empty
- *seq* - to fetch information sequentially from a table

The process of accessing the database using the Tcl API is illustrated in Figure 4.2.

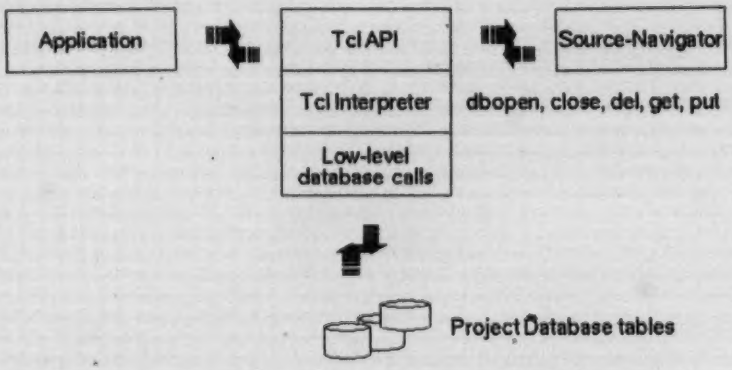

Figure 4.2: Source Navigator Database Access

Type	C/C++	Java	Tcl
cl	class, structure	class	namespace
con	constant	static final	
e	enumeration		
ec	enumeration literal		
fd	function decl.		
fr	friend		
fu	function		procedure
gv	global variable		global variable
iv	variable member	variable member	namespace variable
lv	local variable	local variable	local variable
ma	macro		
md	method decl.		
mi	method impl.	method impl.	namespace procedure
t	typedef		
un	union		

Table 4.1: Source Navigator types used for predefined languages

The Tcl commands for accessing the Source Navigator database are built into the Source Navigator Tcl interpreter called *hyper*, which extends a Tcl interpreter with the new commands. Also, *hyper* contains some variables used in the Source Navigator environment such as the *sn_sep* variable that allows specifying the separator of the fields in the database.

Parsers in Source Navigator are predefined for the following languages: C/C++, Java, Tcl/[incr Tcl], FORTRAN, COBOL, and PowerPC. An extension of Source Navigator called Source Navigator Extensions [SNE] enhances the collection of standard parsers with parsers for several other languages such as: Visual Basic, Jscript, VBScript, HTML, ASP, SQL, and CSS. However, we used only the parsers for C/C++ and Java in our approach. We have tried also the other parsers, but they are far from being complete.

The types used by Source Navigator to represent the facts for the supported languages are provided in Table 4.1. The first column represents the type of the existing facts and denotes the extension of the table file of the database. The other columns describe the meaning of the *generic* types from the first column in different languages. For example, the *cl* type refers to a class in Java, while in Tcl refers to a namespace. The empty cells represent types that are not used in the language for that column.

In addition to the type information presented in Table 4.1, Source Navigator stores the relationship information in two tables with the suffixes *by* and *to*. The *to* table keeps the *refers-to* information and the *by* table the *referred-by* information. In theory these two tables should contain the same information, with the key of the *by* table in reverse of the key of the *to* table. In reality, we found that the *by* table has substantially less information than the *to* table, so that we used only the *to* table to extract the relationships among the artifacts.

Source Navigator database uses also the following tables (where relevant): the *f* table for keeping the files of the project, the *in* table to keep the inheritance relationships among the classes, the *iu* table to keep the includes relationships among the files, and the *fil* table to keep the symbols associated with each files.

4.2 Output Formats

We use two output formats for representing the facts: GXL, and an XML-based format used by the Clare tool. Following, this section presents an overview of the two output formats.

4.2.1 Graph eXchange Language (GXL) Representation

GXL [GXL] has been designed to be a standard exchange format for graph-based tools - implicit for reverse engineering tools that maintain the factbases with the nodes representing the language types (such as function, variable or class), and the edges representing the relationships among the language types (such as uses, friends or inheritance). GXL is designed mainly to facilitate the interoperability of reengineering tools and components such as extractors, analyzers and visualizers. GXL allows software reengineers to combine single-purpose tools especially for parsing, source code extraction, architecture recovery, data flow analysis, pointer analysis, program slicing, query techniques, source code visualization, object recovery, restructuring, refactoring, remodularization and so on, into a single powerful reengineering workbench.

GXL is based on XML (eXtensible Markup Language) [XML], and supports exchanging instances of graphs together with their appropriate schema information. Using GXL with a tool (i.e., importing or exporting GXL files from the tool) provides the power of communication between the tool and the related tools from the same research area. Therefore, more and more reverse engineering tools have started to use GXL as the input format.

To provide interoperability of the graph-based tools, a standard exchange language for graphs has to support a large variety of graph models (e.g. directed graphs, node attributed graphs, edge attributed graphs, node typed graphs, edge typed graphs, ordered graphs, trees, and so on). GXL represents typed, attributed, directed, and ordered graphs. The body of a GXL document is enclosed between <*gxl*> tags. The graph is defined using <*graph*> tags, and a unique identifier is provided for the described graph. At the next level, the nodes and edges are listed using <*node*> and <*edge*> tags, respectively. Every node or edge has a unique id, a type pointing to the defined schema information (using <*type*> tags), and attributes (using <*attr*> tags). GXL provides primitive types for the values of the attributes such as *bool*, *int*, *float*, or *string*, and complex types specified as *enum*, *seq*, *set*, *bag*, *tup*,

or *locator* (URI-references to externally stored objects). The direction of the edge is given by the *to* and *from* attributes associated with the edge.

Figure 4.3 presents a simple GXL instance file. The graph *example* is defined in this GXL. It contains two nodes having the ids *f1* and *v1*, with the associated names *main* and *i*. *main* has the type *function*, while *i* has the type *variable*. Both of them have an attribute *file* with the value *main.c*. The graph defines also an edge from *f1* to *v1* of type *uses* and the attribute *line* equals to 127. In reverse engineering terms, this example illustrated two node artifacts, a function and a variable, and the relationship between them (the *main* function *uses* the *i* variable at line 127).

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE gxl SYSTEM "http://www.gupro.de/GXL/gxl-1.0.dtd">
<gxl>
  <graph id="example">
    <node id="f1" name="main">
      <type name="function" />
      <attr name="file">
        <string>main.c</string>
      </attr>
    </node>
    <node id="v1" name="i">
      <type name="variable"/>
      <attr name="file">
        <string>main.c</string>
      </attr>
    </node>
    <edge id="e1" to="v1" from="f1">
      <type name="uses"/>
      <attr name="line">
        <int>127</int>
      </attr>
    </edge>
  </graph>
</gxl>
```

Figure 4.3: GXL representation example

The schema that contains as nodes the language element types (such as function, variable, and class) and as edges the relations among the language element types (such as calls, uses, or defines) is a graph itself. Therefore, the schema can be represented also using GXL, in the same way the instance graph of the facts is represented. GXL provides a tool to check if the instance graph conforms to the corresponding schema.

4.2.2 Clare XML-based Representation

GXL is a nice way to represent any general graph. However, the generality of the GXL representation has some drawbacks. One of the GXL drawbacks is that when representing factbases the size of the instance file is very big. The size drawback implies a long time of loading the instance file.

Therefore, we develop another XML-based representation for the factbases, mostly to improve the efficiency of the Clare tool. We will refer to this representation of the factbases as the Clare representation. This representation is more compact then the GXL representation. Thus, the factbase file is smaller in size than the corresponding GXL representation, while maintaining all information.

We describe the Clare representation based on an example. Let's consider the GXL factbase illustrated in Figure 4.3. The corresponding Clare representation given in Figure 4.4. We practically eliminated one level presented in the GXL representation given by the elements of a general graph (such as node, edge, or attr). In the Clare representation the XML nodes become the type of the fact from the GXL instance representation. The XML attributes in Clare format are the GXL attribute nodes.

```
<?xml version="1.0" encoding="UTF-8"?>
<example>
  <function id="f1" name="main" file="main.c">
  <variable id="v1" name="i" file="main.c">
  <uses id="e1" from="f1" to="v1" line="127">
</example>
```

Figure 4.4: Clare representation example

4.3 Extracting the Facts

This section presents our approach for extracting the facts from a software system in GXL and Clare representations using Source Navigator as the front-end.

Our approach consists of two major steps. In a first step the Source Navigator is used as a front-end to extract information from the subject software system and to populate its internal database. In the second step, we developed several Tcl scripts to extract the factbase into the two aforementioned representations. The next two subsections present more details about each step.

4.3.1 Populating Source Navigator Internal Database

In this step, Source Navigator is used to populate its internal database. We set the filename extensions for the files that will be associated with a Source Navigator parser. There are by default some predefined extensions for each parser. For example, for the Tcl parser the default extensions of the files that will be parsed using this parser are: *.tcl*, *.itcl*, *.itk*, *.tk*, and *.test*. However, sometimes we need additional extensions of files to be considered. For example, suppose that a *.rcl* extension denotes a Tcl file (i.e., the file contains a Tcl script), then we associate the *.rcl* extension with the Tcl parser. To accomplish this setting, we modify the *%HOME_SN%/snavigator/etc/sn_prop.cfg* file. In this file, a line denotes the configuration for the Tcl parser in Source Navigator environment (the line starts with *sn_add_parser tcl*). A parameter of the command, called *suffix*, contains the default extension for the files to be parsed using the Tcl parser. We can modify this parameter to contain the new extensions to be supported by the parser. The same settings may be operated for each parser supported by the Source Navigator.

After setting up the configuration for each parser, we create a new project that contains the source code information of the input software system. There are two ways to accomplish this: using the Source Navigator user interface or from the command line. We chose the Source Navigator user interface mode because it was easier to add and remove files for a project. For creating a project in Source Navigator interface, we provide the project file name and the directories that contain the source code for the subject system. A more detailed selection of the files could be managed using the Project Editor that allows to add/remove files to the project. Then, in a first step, Source Navigator invokes for each file the parser associated with its extension, and produces the facts existing in these files. In a second step, Source Navigator builds the relationship tables based on the information extracted previously. By default, the table files are stored under the *.snprj* directory that is built automatically in the same directory where the project file name resides.

Table 4.2 shows the Source Navigator internal database format for the records of different fact tables. A record contains two parts, the key and the data part, separated by a semicolon character (;). The symbol *?* represents the *sn_sep* separator character, used to separate the fields in the key and the data part. Positions consist of line and column numbers separated by a comma (,). The hash character (#) in class names means that the symbol does not belong to any classes. The class table has three fields for the key (*name*, *start position*, and *file name*), and six fields for the data part (*end position*, *kind*, *class template*, *comment*, and two unused fields).

In the following step, we use the Source Navigator database, populated with the facts from the system, to produce the factbase file in two different representations discussed before.

Table	Description	Record Format
by	referred by	ref-class?ref-symbol-name?ref-type?class?symbol?type?access?position?filename;{caller_argument_types}?{ref_argument_types}
cl	classes	name?start_position?filename;end_position?attributes?{}?{class_template}?{}?{comment}
con	constants	name?start_position?filename;end_position?attributes?{dec_type}?{}?{}?{comment}
e	enumerations	name?start_position?filename;end_position?attributes?{}?{}?{}?{comment}
ec	enumeration constants	name?start_position?filename;end_position?attributes?{}?{}?{}?{comment}
f	project files	name;group?parsing-time?highlight-file
fd	function	name?start_position?filename;end_position?attributes?{ret_type}?{arg_types}?{arg_names}?{comment}
fil	symbols of files	filename?start_position?class?identifier?type;end_position?high_start_pos?high_end_pos?arg_types
fr	friends	name?start_position?filename;end_position?attributes?{ret_type}?{arg_types}?{arg_names}?{comment}
fu	functions	name?start_position?filename;end_position?attributes?{ret_type}?{arg_types}?{arg_names}?{comment}
gv	variables	name?position?filename;attributes?{type}?{template?parameter}?{comment}
in	inheritances	class?base-class?start_position?filename;end_position?attributes?{}?{class template}?{inheritance?template}?{comment}
iu	include	included_file?start_position?include_from_file;0.0?0x0?{}?{}?{}?{}
iv	instance variables	class?variable-name?start_position?filename;end_position?attributes?{type}?{}?{}?{comment}
lv	local variables	function?variable-name?start_position?filename;end_position?attributes?{}?{type}?{}?{comment}
ma	macros	name?start_position?filename;end_position?attributes?{}?{}?{}?{coimment}
md	method definitions	class?name?start_position?filename;end_position?attributes?{ret_type}?{arg_types}?{arg_names}?{comment}
mi	method implementations	class?name?start_position?filename;end_position?attributes?{ret_type}?{arg_types}?{arg_names}?{comment}
t	typedefs	name?position?filename;attributes?{original}?{}?{comment}
to	refers to	class?symbol-name?type?ref-class?ref-symbol?ref-type?access?position?filename;{caller_argument_types}?{ref_argument_types}
un	unions	name?position?filename;attributes?{}?{}?{comment}

Table 4.2: Source Navigator internal database format

4.3.2 Producing the Output Factbase

In this step, the Source Navigator database is queried using some Tcl scripts to extract the facts in GXL and Clare representations.

The idea of the extractor is to traverse the Source Navigator tables produced in the previous step, and to write the facts in the needed formats.

Figure 4.5 presents a snippet of script code from the extractor implementation. The script takes the following input parameters: the directory where the Source Navigator project resides, the name of the Source Navigator project, the format of the output factbase, and optionally the name of the output factbase file. If the name of the output factbase file is not provided, then the script displays the output factbase using the standard output. The *run_extractor* Tcl procedure is the main entry point of the extractor. It interrogates the Source Navigator database associated with the subject software system by going over the source files, other node types (such as classes or variables), *include*, *friend*, and *inheritance* relationships, and *refers to* relationship table. The *retrieve_files* Tcl procedure retrieves the facts based on the *file* Source Navigator table. The snippet of code shows how to iterate sequentially over the *file* Source Navigator table. A similar procedure is used to interrogate the other entity types.

Overview Extraction Approach

Our scripts retrieve the source files existing in the subject project from the *f* table (*retrieve_files* procedure). Each source file is represented by a node in the resulting factbase file having an attribute called *language* that contains the programming language used in the source file. Then, the scripts extract the symbol facts from all the source files from the *fil* table. The *fil* table does not contain all the attributes for the symbols. Thus, for each symbol, the script looks in the table associated with its type for retrieving all the attributes for that symbol. The symbol and its attributes are then written in the output factbase file.

Following, our approach extracts the relationships among the facts inside each language. The include relationships among files are extracted from the *iu* table. Then, the friend and inheritance relationships (if they exist for the subject system) are extracted from the *fr* and *in* table. Finally, the script retrieves the main part of the relationships from the *to* table, which contains the referred to relations. Based on the fact types involved in the relation, we can induce the type of the relation, which is not specified in the *to* table. For example, if the facts involved in a relation have the type *function*, then the relation type is *calls*.

Generating Unique Ids for Facts

The output factbase uses unique ids to identify the nodes and edges over the factbase. For edges, it is easy to generate ids in this way: every edge will have a number id increased every time a new edge occurs. This is reasonable

```
if {${argc} < 3 || ${argc} > 4} {
    puts "Usage: ${argv0} projectdir projectname ";
    puts "[GXL|Clare] [outputfile]";
    exit;
}

#create the factbase for a given project
proc run_extractor {} {
    . . .
    retrieve_files;
    retrieve_artifact_nodes;

    retrieve_include_rels;
    retrieve_friends_rels;
    retrieve_inheritance_rels;
    retrieve_to_rels;
    . . .
}

#retrieve the files from the SN project (from file .f)
proc retrieve_files {} {
    global db;
    global out_fd;
    global node_artifacts;
    global type_artifacts;

    if {![info exists db(f)]} {
        print_message "Error! The .f file does not exist!";
        return;
    }
    foreach def [$db(f) seq] {
        set filename "[lindex $def 0]";
        set language "[lindex $def 1]";
        set language "[lindex $language 0]";

        set node_artifacts($filename) "$filename";
        set type_artifacts($filename) "f";

        catch { unset attrs; }
        set attrs(language) "$language";

        write_node $out_fd $filename attrs;
    }
}
```

Figure 4.5: Snippet from the extractor implementation

because we do not need to make connections among the edges. On the other hand, for the nodes, we build the associated ids based on the name of the fact, the source file in which the fact is defined, and the line and column at which the fact is defined (except the file node types, for which the id is based on the name of the file including its path). The constituent nodes of an edge are given as ids.

In some situations we have to identify the same fact among different tables. For example, this case appears when the scripts extract the "referred to" relationships. The index of the *to* table consists of the following information: class, name, type, referred class, name referred, type referred facts. The data presented in the *to* table contains the argument types for both the fact and the referred fact (only in the case of functions and methods). Based on this information, we have to be able to recuperate the id associated with the facts. We query the table associated with the type of the fact to retrieve the file name, column and line of the fact to be used to build the id. A problem appears when overloading a function or method in C++ or Java languages. In this case, we have to apply some heuristics to determine the right fact in order to use the correct id in the output factbase file.

Some example of heuristics we used to determine the match of a function or method among different tables are:

- match using simply the name of the function/method
- if there are more functions/methods matched, then compare their parameter types
- if there are still more candidate functions/methods, then match the source file of the candidate functions/methods with the source file where the relation appeared;
- if there are still more candidates, then take the first match from the list of candidates

4.4 C/C++ and Java Schemas

A *domain model* represents the structure of the entities, relations, and attributes in a specific area. It represents the meta-data of information in an application. The *domain model* and *schema* have the same meaning in the reverse engineering area. Three schemas for GXL are the Datrix schema [Holt00], the Columbus schema [Ferenc01, Ferenc02], and Dagstuhl Middle Model (DMM) [DMM], which represent software graphs. DMM itself is documented with a UML class diagram of the nodes and relationships that may appear in a GXL file.

The schema for source code facts may be established fully before developing the fact extractor. In practice, however, there is an iterative process of refining

the schema and fact extractor together for the needs of analyzing the source code of a software system.

The schemas for the extractors are based on what fact types the Source Navigator parsers produce. Therefore, the types from Table 4.2 appear as node types in our schema. The attributes are mainly based on the fields existing in a record from Source Navigator tables. Source Navigator does not provide the name of the parameters for a function or method. We extracted this information going directly into the source file and doing a local search for the function/method involved. We will use the name of the parameters of a function/method in the extraction algorithm of the cross-language dependencies.

Following, we present two schemas for C/C++ and Java programming languages.

C/C++ Schema

Our C/C++ schema is based on the elements presented in the C/C++ programming languages. Table 4.3 presents the node and relationship types for our C/C++ schema. The C/C++ extractor generates the factbase based on our C/C++ schema. The first column in the table represents the node types presented in a C/C++ factbase (e.g., *Class*, *Function*, or *GlobalVar*). The first row in the table denotes the relation types between the node types (e.g., *uses* or *calls*). The other cells in the table represents the destination node type of the relation type from the appropriate column header. Therefore, an entry in the table is read in the following way: node type from the first column, relation type from the first row, node type from the intersection between the node type row and relation type column. For example, the following relation types are in our C/C++ schema: "*Method calls Function*", "*Method uses LocalVar*", or "*Class has MemberVar*".

Table 4.4 presents the attribute types for both node and relation types from our C/C++ schema. First column from the table contains all the node and relation types from the C/C++ schema. The attribute types associated with the node or relation type from the first column is given in column two. For example, the *File* node type has the following attribute types: *name* (the name of the file without the path), *sourcefile* (the complete name of the file - including the path), and *language* (the programming language associated with this source file). If an attribute type has only certain values, then the possible values are given in italic font after the attribute. For example, the *kind* attribute type for a *Function* node type might have the value *inline* when the function is defined as inline. If more than one value are presented for the *kind* attribute type, then the values are separated by a comma character.

	calls	friend of	has	includes	inherits from	uses
Class		Class	MemberVar Method MethodDecl		Class	Class Union
Enum			EnumValue			
EnumValue						
File				File		
Function	Function FunctionDecl Method MethodDecl	Class				Class Enum EnumValue GlobalVar LocalVar Macro MemberVar Union
FunctionDecl						
GlobalVar						
LocalVar						
Macro						
MemberVar						
Method	Function FunctionDecl Method MethodDecl					Class Enum EnumValue GlobalVar LocalVar Macro MemberVar Union
MethodDecl						
Typedef						
Union			MemberVar			Union

Table 4.3: C/C++ Schema - Node and Relation Types

Node & Relation Types	Attribute Types
Class	name, sourcefile, line, column, endline, endcolumn, kind (*structdef*)
Enum	name, sourcefile, line, column, endline, endcolumn
EnumValue	name, sourcefile, line, column, endline, endcolumn
File	name, sourcefile, language
Function	name, sourcefile, return_type, arg_names, arg_types, line, column, endline, endcolumn, kind (*inline*)
FunctionDecl	name, return_type, arg_names, arg_types
GlobalVar	name, sourcefile, real_type, line, column, endline, endcolumn
LocalVar	name, sourcefile, real_type, line, column, endline, endcolumn
Macro	name, sourcefile, line, column, endline, endcolumn
MemberVar	name, sourcefile, real_type, line, column, endline, endcolumn, kind (*private*, *protected*, *public*)
Method	name, sourcefile, return_type, arg_names, arg_types, line, column, endline, endcolumn, kind (*private*, *protected*, *public*, *virtual*, *inline*, *constructor*, *destructor*, *purevirtual*)
MethodDecl	name, return_type, arg_names, arg_types
Typedef	name, sourcefile, real_type, line, column, endline, endcolumn
Union	name, sourcefile, line, column, endline, endcolumn
calls	sourcefile, line, column, callee_args
friend_of	sourcefile, line, column
has	sourcefile, line, column
includes	sourcefile, line, column
inherits_from	sourcefile, line, column
uses	sourcefile, line, column

Table 4.4: C/C++ Schema - Attribute Types

	calls	has	inherits from	uses
Class		MemberVar Method	Class	Class
File				
LocalVar				
MemberVar				
Method	Method			Class Enum LocalVar MemberVar

Table 4.5: Java Schema

Node & Relation Types	Attribute Types
Class	name, sourcefile, line, column, endline, endcolumn, kind (*abstract* , *interface*)
File	name, sourcefile, language
LocalVar	name, sourcefile, real_type, line, column, endline, endcolumn
MemberVar	name, sourcefile, real_type, line, column, endline, endcolumn, kind (*private* , *protected* , *public* , *static* , *final*)
Method	name, sourcefile, return_type, arg_names, arg_types, line, column, endline, endcolumn, kind (*private* , *protected* , *public* , *static* , *native* , *synchronized* , *constructor* , *destructor*)
calls	sourcefile, line, column, callee_args
has	sourcefile, line, column
inherits from	sourcefile, line, column
uses	sourcefile, line, column

Table 4.6: Java Schema - Attribute Types

Java Schema

Our Java schema is based on the elements presented in the Java programming language. Table 4.5 presents the node and relationship types for our Java schema. The Java extractor generates the factbase based on our Java schema.

Table 4.6 presents the attribute types for both node and relation types from our Java schema.

Note that C/C++ and Java schemas have commonalities. We will describe later in this thesis how we combined the schemas for different languages to obtain a common schema. A common schema helps to parse easily any factbase containing facts from a programming language represented in the common schema.

4.5 C/C++ and Java Results

This section presents the results of the extractors based on Source Navigator on some simple examples. The results of the extractors are presented in the Clare representation. The GXL representation of the results is omitted because of the size of the GXL file.

4.5.1 C/C++ Factbase Example

Consider the C++ snippet of source code from Figure 4.6. The *Foo* C++ class is declared in the *Foo.h* file, and defined the *Foo.cpp* file. The class *Foo* contains the *m_x* private member variable of type *int*, two constructors, one destructor, and the *Pow* method. The constructors and the destructor are defined in the header file, while the *Pow* method is declared in the *Foo.cpp* file.

Foo.h :

```
class Foo {
   int m_x;
public:
   Foo() { m_x = 0; };
   Foo(int x) { m_x = x; };
   virtual ~Foo() { };
   int Pow();
};
```

Foo.cpp :

```
#include "Foo.h"
int Foo::Pow(){
   return m_x * m_x;
}
```

Figure 4.6: A simple C++ example

The result of the extractor for this simple C++ example is illustrated in Figure 4.7.

4.5.2 Java Factbase Example

Consider the Java snippet of source code from Figure 4.8, which is the correspondence of the C++ code from Figure 4.6. The *Foo* Java class is declared and defined in the *Foo.java* file. The class *Foo* contains the x private member variable of type *int*, two constructors, and the *Pow* method.

The result of the extractor for this simple Java example is illustrated in Figure 4.9.

4.6 Summary

This chapter presented a method to build extractors using Source Navigator as a front-end. The C/C++ and Java extractors are part of the fact extraction component in our architecture. We used the Source Navigator tables to extract the facts and relationships in two XML-based representations. We have introduced in this chapter two schemas for artifacts of interest in C/C++ and Java

```
<?xml version="1.0" encoding="ASCII" ?>
<cpp xmlns="http://www.cppfactbase.org">
 <File id="n1" language="c++" name="Foo.cpp" />
 <File id="n2" language="c++" name="Foo.h" />
 <Method id="n3" endcolumn="1" endline="5" sourcefile="Foo.cpp"
  name="Pow" returntype="int" column="9" line="3" />
 <Class id="n4" endcolumn="1" endline="8" sourcefile="Foo.h"
  name="Foo" column="6" line="1" />
 <MemberVar id="n5" kind="private" endcolumn="8" endline="2"
  sourcefile="Foo.h" name="m_x" realtype="int" column="5" line="2" />
 <Method id="n6" kind="constructor public" endcolumn="19"
  endline="4" sourcefile="Foo.h" name="Foo" column="1" line="4" />
 <Method id="n7" argnames="x" argtypes="int"
  kind="constructor public" endcolumn="24" endline="5"
  sourcefile="Foo.h" name="Foo" column="1" line="5" />
 <Method id="n8" kind="destructor public virtual" endcolumn="19"
  endline="6" sourcefile="Foo.h" name="~Foo" column="9" line="6" />
 <MethodDecl id="n9" sourcefile="Foo.h" name="Pow" column="5" line="7" />
 <includes from="n1" to="n2" line="1" />
 <has from="n4" to="n5" sourcefile="Foo.h" line="2" />
 <has from="n4" to="n6" sourcefile="Foo.h" line="4" />
 <has from="n4" to="n7" sourcefile="Foo.h" line="5" />
 <has from="n4" to="n8" sourcefile="Foo.h" line="6" />
 <has from="n4" to="n9" sourcefile="Foo.h" line="7" />
 <has from="n4" to="n3" sourcefile="Foo.cpp" line="3" />
 <uses from="n6" to="n5" sourcefile="Foo.h" line="4" />
 <uses from="n7" to="n5" sourcefile="Foo.h" line="5" />
 <uses from="n3" to="n5" sourcefile="Foo.cpp" line="4" />
</cpp>
```

Figure 4.7: Factbase for the C++ example in Clare representation

Foo.java :

```
public class Foo {
   private int x;
   public Foo() { x = 0; };
   public Foo(int x1) { x = x1; };
   public int Pow(){
      return x * x;
   };
}
```

Figure 4.8: A simple Java example

```
<?xml version="1.0" encoding="ASCII" ?>
<java xmlns="http://www.javafactbase.org">
 <File id="n1" language="java" name="Foo.java" />
 <Class id="n2" kind="public" endcolumn="1" endline="8"
  sourcefile="Foo.java" name="Foo" column="13" line="1" />
 <MemberVar id="n3" kind="private" endcolumn="16" endline="2"
  sourcefile="Foo.java" name="x" column="15" line="2" />
 <Method id="n4" kind="constructor public" endcolumn="26" endline="3"
  sourcefile="Foo.java" name="Foo" column="10" line="3" />
 <Method id="n5" argnames="x1" argtypes="int"
  kind="constructor public" endcolumn="33" endline="4"
  sourcefile="Foo.java" name="Foo" column="10" line="4" />
 <Method id="n6" kind="public" endcolumn="4" endline="7"
  sourcefile="Foo.java" name="Pow" returntype="int" column="14" line="5" />
 <has from="n2" to="n3" sourcefile="Foo.java" line="2" />
 <has from="n2" to="n4" sourcefile="Foo.java" line="3" />
 <has from="n2" to="n5" sourcefile="Foo.java" line="4" />
 <has from="n2" to="n6" sourcefile="Foo.java" line="5" />
 <uses from="n4" to="n3" sourcefile="Foo.java" line="3" />
 <uses from="n5" to="n3" sourcefile="Foo.java" line="4" />
 <uses from="n6" to="n3" sourcefile="Foo.java" line="6" />
</java>
```

Figure 4.9: Factbase for the Java example in Clare representation

programming languages. The schemas are based on the elements recognized by the Source Navigator extractors. Some simple examples of the extractor results are illustrated.

Next chapter presents another method to build an extractor, by hooking into an implementation of the language. The main focus of this method is to handle scripting languages.

Chapter 5

Extracting Facts for Perl

Scripting and shell languages are a highly flexible way to integrate functionality in a software system. For example, scripts can prepare input data, sequence the running of programs, consolidate outputs, wrap components, or implement functionality themselves. Scripts may serve as crucial glue code to tie together the diverse technologies used in a heterogeneous system. Scripts are generally easy to write and run. A programmer does not typically need to declare the types of variables, or deal with separate binary object code. Thus, scripting languages and their support environments are well suited for rapid software development.

Scripting languages are often dynamic in nature, and the line is often blurred between what is data and what is code. For example, in Perl [Parrot, PerlBible, PERL], one can put the text of a subroutine definition in a string, and have that string evaluated to define temporarily a new subroutine at run time. Furthermore, languages like Perl have evolved over time to have rich behaviors and idioms, much like a natural language. There may be many different ways to express the same intended effect, and the meaning of something may depend highly on the context. These aspects can create problems in understanding Perl code, especially for non-experts.

Software reverse engineering research has largely focused on traditional programming languages, such as C, C++, and Java. These programming models are well understood, and many fact extractors and analyzers exist to discover the structure of code in these languages [Dean01, Columbus, Korn99, Moise03, Singer97, Tilley94]. There has not been as much attention on scripting languages, such as Perl. To create a fact extractor for Perl code, one could conceivably start by writing a lexer to scan for relevant constructs, or write a parser to build some form of abstract syntax tree for analysis. Still, one needs to be very fluent in Perl to develop a correct extractor for all the constructs and subtle idioms. For example, Perl allows the same name to be used again for entities in a package, as long as they differ in type.

The approach we take is to use the implementation of the Perl interpreter itself to extract facts for Perl code, since the interpreter is authoritative (mod-

ulo any defects) for the meaning of some Perl construct. We are not writing a Perl fact extractor as a Perl script. Rather, we developed an extraction component that refers to internal data structures of the Perl interpreter, which are populated appropriately just before running a Perl script. There is a single insertion point where this component is invoked to generate facts about the code of the Perl script. Since the script is loaded, but not actually run, the technique is static.

The tradeoff here is that the internal data structures of the Perl interpreter need to be well understood, and there is a deep dependence by the extraction component on these data structures. Thus, much program comprehension of the Perl interpreter is involved. Still, there is a huge benefit in reusing the interpreter, rather than redundantly writing our own front end and program representation. The idea is to hook into the interpreter at the latest possible moment, just before script execution, to reuse as much of the interpreter and its code analysis as possible, yet maintain a static technique. We believe a similar technique would work to extract facts for other programming or scripting languages implemented through an interpreter.

As related work, the approach is somewhat akin to the CPPX fact extractor [Dean01] for C++, which leverages the GNU gcc compiler. Development environments for Perl do exist [Epic, Komodo, Affrus], such as Affrus and Komado, but they are aimed toward code editing and debugging, not to create facts for software reverse engineering and visualization tools.

The extracted facts conform to a schema that is represented in GXL and Clare XML-based formats. A test evaluates how the extractor processes all the Perl modules provided with the Perl distribution source code.

In this chapter, Section 5.1 details the discovered internal data structures and interpretation process of the Perl interpreter (for which the existing documentation is scattered and poor). Section 5.2 builds upon this understanding to describe the Perl fact extractor component, and how it consults the data structures to generate the appropriate facts. Section 5.3 describes the schema for the facts emitted by the component, and Section 5.4 highlights the results of a test. Section 5.5 concludes the chapter, and outlines directions for future work.

5.1 Perl Internals

This section presents the major internal data structures in the Perl interpreter that represent Perl language elements. Our extractor interrogates the Perl internals just before the execution phase of the Perl script. At that moment the internal data structures are ready to be used for fact extraction.

Perl is a compiling interpreter. Instead of interpreting line-by-line the script file, it reads the entire script file, converts it to an internal representation, and then executes the instructions. This process is similar to the JVM (Java

Virtual Machine), except that the set of basic instructions is different. There are two major phases used to interpret a Perl script.

- **Parsing and Compiling.** The script is parsed and converted into operation trees, one tree for each subroutine. This phase also involves an optimization process for the operation trees (such as constant propagation). Each node in an optimized operation tree is linked to the next node in execution order.
- **Executing.** Traverse over the operation trees and execute the operation in each node.

The Perl interpreter is written entirely in C language. We used Perl version 5.8.6, which contains about 150 KLOC among 90 core files.

The comprehension of the source code of the Perl interpreter is a difficult process mainly because of the overuse of macro-definitions. That is, macro names do not appear while debugging the source code to comprehend the flow of the interpreter. The macro definitions are used for various reasons such as:

- to maintain the compatibility of the Perl API for different versions of Perl;
- to provide an abstraction layer for different C functions which are not compatible for different platforms; and
- to hide the arguments used in almost all the calls (e.g., the Perl interpreter structure is a global variable, and a pointer to it is passed in almost every call).

Three data types cover most of the data that are manipulated in a Perl script: *scalar* (integer, floating point number, text string, or reference), *array*, and *hash*. Perl also offers other fundamental types: *filehandle* (handle to an open file or directory, including state information), *typeglob* or *glob* (composite data type of all the other types), and *undefined* (undefined value).

The following snippet of Perl code presents some examples of the fundamental types. Here, i contains an integer scalar; f contains a floating-point scalar; s contains a string; r contains a reference to i; h contains a hashtable with two keys *key1* and *key2* that have associated values *value1* and *value2*, respectively; a contains an array of values *1*, *2*, *3*; *FILE* is a handle to the file named *filename*; g is a glob that contains both an integer scalar and an array; and u is an undefined variable. There is no type associated with u, although internally it is a scalar.

```
$i = 42;
$f = 1.01;
$s = "text";
$r = \$i;
%h = ('key1' => 'value1',
      'key2' => 'value2');
@a = (1, 2, 3);
open FILE, 'filename';
*g = \$i;
*g = \@a;
$u;
```

To distinguish the various types internally, Perl uses type flags (C enumeration literals of the *svtype* enumerated type). Corresponding to these type flags are C structure types that contain pointers to structures for the actual data (see Table 5.1). These C structure types (e.g., *SV*, *AV*, *HV*) all contain only 3 fields (named *sv_any*, *sv_refcnt*, and *sv_flags*), which have compatible types to allow structural assignment among these structures. The actual data is stored in a structure of a certain type, and there is a type extension hierarchy among these structures (see Figure 5.1). Subtypes in the hierarchy add one or more fields to a supertype.

Data Type	Type Flag	Structure	Actual Data
Undefined	SVt_NULL	SV	NULL
Reference	SVt_RV	SV	XRV
String	SVt_PV	SV	XPV
Integer	SVt_IV	SV	XPVIV
Floating-point	SVt_NV	SV	XPVNV
Object/Tie	SVt_PVMG	SV	XPVMG
Array	SVt_PVAV	AV	XPVAV
Hashtable	SVt_PVHV	HV	XPVHV
Glob	SVt_PVGV	GV	XPVGV
Subroutine	SVt_PVCV	CV	XPVCV
Filehandle	SVt_PVIO	IO	XPVIO
Format	SVt_PVFM	SV	XPVFM

Table 5.1: Perl data type correspondence

5.1.1 Perl Data Types

Scalars

The *SV* C structure is used as the base structure for any Perl type. It has three fields: a void pointer called *sv_any*, which points to the structure for the actual data; a reference counter called *sv_refcnt*, which tells when to destroy the scalar; and an unsigned 32 bit integer called *sv_flags*, which keeps the type flag in the first 8 bits and uses the remaining 24 bits to encode how various

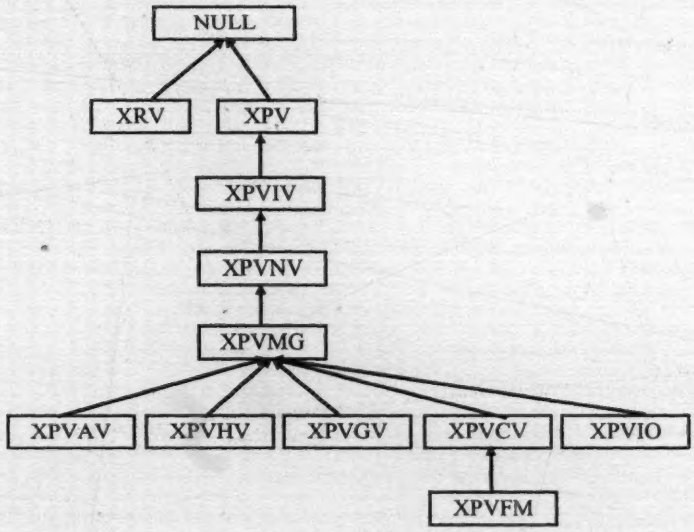

Figure 5.1: Internal Perl data type hierarchy

field values of the actual data should be interpreted. The definition of the *SV* structure and the associated macros to get and set its fields are found in the file *sv.h*. For example, the *SvTYPE(SV*)* macro is used to obtain the type flag from an *SV* structure.

The *SV* structure is general, since the *sv_any* void pointer field could point to any kind of structure containing the actual data. Thus, any Perl data type can be represented through an *SV* structure. Perl consults the *sv_flags* field for the appropriate structure type pointer for typecasting the *sv_any* pointer.

An undefined type has the type flag equal to *SVt_NULL*. The *sv_any* field of the *SV* structure has the value NULL for an undefined type.

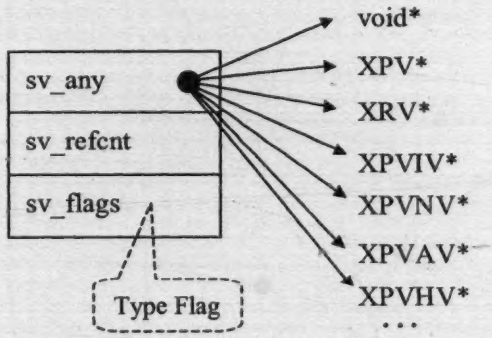

Figure 5.2: General Perl *SV* structure

For a Perl reference, the *SVt_RV* type flag is used, and the *SV* structure has *sv_any* pointing to an *XRV* structure. This structure contains a single field

called *xrv_rv*, which points to a general *SV* structure (or compatible type).

A Perl scalar that holds a string value has the underlying type flag *SVt_PV*, and the *SV* structure has *sv_any* pointing to a structure of type *XPV*.

XPV fields:

- *xpv_pv* - pointer to allocated memory
- *xpv_cur* - length of xpv_pv as a C string
- *xpv_len* - allocated memory size

A Perl integer scalar has the type flag *SVt_IV*, and the *SV* structure has *sv_any* pointing to an *XPVIV* structure to store the actual integer. A bit setting in *sv_flags* denotes whether a value stored is valid or not. The *XPVIV* structure contains, besides the fields of *XPV*, a field called *xiv_ivx* to keep the integer value.

Similar to an integer scalar, a Perl floating-point scalar has the type flag *SVt_NV*, and the *SV* structure has *sv_any* pointing to an *XPVNV* structure to store the actual floating-point number. The *XPVNV* structure contains, besides the fields of *XPVIV*, a field called *xnv_nvx* to keep the floating-point value.

Hashtables and Arrays

For Perl objects and/or ties, the *SVt_PVMG* type flag is used. A *tie* has a special *SV* structure that triggers different actions (routines) when the scalar is touched (via a get, set, clear, or free operation). Perl ties are usually used when implementing database applications. For the *SVt_PVMG* type flag, the *sv_any* field points to an *XPVMG* structure. The *XPVMG* structure has all the fields of the *XPVNV* structure, plus two more fields.

XPVMG selected fields:

- *xmg_magic* - linked list of magicalness
- *xmg_stash* - object package

The Perl array data type is identified by the type flag *SVt_PVAV*. The associated structure has type *AV* (assignment compatible with *SV*) with *sv_any* pointing to an *XPVAV* structure.

XPVAV selected fields:

- *xav_array* - pointer to first array element
- *xav_fill* - index of last element present
- *xav_max* - max index of array
- *xav_alloc* - pointer to allocated array of SVs

The Perl hashtable data type is identified by the type flag *SVt_PVHV*. The associated structure has type *HV* (assignment compatible with *SV*) with *sv_any* pointing to an *XPVHV* structure. Perl hashtables keep the key/value pairs in an *HE* structure, and the keys in an *HEK* structure.

XPVHV selected fields:

- *xhv_array* - pointer to allocated array

- *xhv_fill* - index of last element present
- *xhv_max* - max index of array
- *xhv_keys* - number of elements in the array
- *xhv_riter* - current root of iterator
- *xhv_eiter* - current entry of iterator
- *xhv_pmroot* - list of modules for this package
- *xhv_name* - name, if a symbol table

The implementation of arrays or hashtables is found in *av.{h,c}* files or *hv.{h,c}*, respectively.

Glob Values
Glob or typeglob values (or "symbols") have the type flag *SVt_PVGV*, and the *sv_any* field points to an *XPVGV* structure. This structure has a field named *xgv_gp*, which keeps a pointer to a *GP* structure that contains pointers to data of various kinds: scalar, array, hashtable, routine, filehandle, and format. The name of the symbol is given by the field *xgv_name* in *XPVGV*. Perl uses a pointer to *GP* instead of including the *GP* structure inside *XPVGV* because a *GP* structure can be shared by one or more glob values.

5.1.2 Perl Subroutines

A Perl subroutine is maintained using a CV (Code Value) structure (assignment compatible with *SV*), with *sv_any* pointing to an *XPVCV* structure.

XPVCV selected fields:

- *xcv_padlist* - subroutine scratchpads
- *xcv_stash* - subroutine stash
- *xcv_xsub* - pointer to a C function for an external subroutine (otherwise NULL for a subroutine defined in Perl)
- *xcv_start* - pointer to subroutine operation tree

Each subroutine has a scratchpad list, which initially contains an array of variable names and an array of literals used in the routine. At runtime, when a subroutine is called, a scratchpad array is added to the list to store the values of lexical variables (declared with *my* in Perl).

Each subroutine has an address to its operation tree. There are 351 predefined operations that can be used in an operation tree. There are 11 different operation structures. For example, the operation node for scalar assignment (*sassign*), is an instance of the *BINOP* operation structure. The Perl predefined operations appear in the *opcode.h* file, and each has an associated C function to implement it. Table 5.2 presents some examples of the Perl operations, their descriptions, and their C function implementions. The Perl operations are defined in the files: *pp.{h,c}*, *pp_ctl.c*, *pp_hot.c*, *pp_pack*, *pp_sort*, and *pp_sys*. There is the notion of a *ppcode*, which is a Perl-stack-based function that implements an operation.

Perl Operator	Description	C function
leavesub	subroutine exit	Perl_pp_leavesub
lineseq	line sequence	Perl_pp_lineseq
nextstate	next statement	Perl_pp_nextstate
sassign	scalar assignment	Perl_pp_sassign
shift	shift	Perl_pp_shift
rv2av	array dereference	Perl_pp_rv2av
rv2sv	scalar dereference	Perl_pp_rv2sv
gv	glob value	Perl_pp_gv
padsv	private variable	Perl_pp_padsv
print	print	Perl_pp_print
pushmark	pushmark	Perl_pp_pushmark
concat	concatenation	Perl_pp_concat
null	null operation	Perl_pp_null
gvsv	scalar variable	Perl_pp_gvsv

Table 5.2: Selected Perl operations

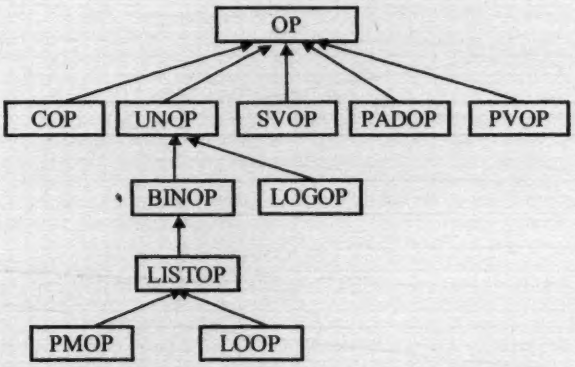

Figure 5.3: Perl operation structure hierarchy

Similar to Perl data types, the 11 Perl operation structures are related in a type extension hierarchy (see Figure 5.3). The *OP* structure, which is the "base" type, contains the following fields:

- *op_next* points to the next *OP* structure to execute after the current one;
- *op_sibling* points to the sibling *OP* structure;
- *op_ppaddr* points to the *ppcode* function for this operation;
- *op_targ* keeps the offset in a scratchpad array;
- *op_type* has the type of the operation;
- *op_seq* contains the execution number assigned by the optimizer at compile time;
- *op_flags* stores the flags common for all operations.

To access a lexical variable, an operation node maintains in the *op_targ* field an offset in the scratchpad of the subroutine where the name of variable resides.

The other operation structures add further fields. For example, the *LISTOP* operation structure has two more fields that point to the first and last operation in a list of operation structures.

Consider the Perl script from Figure 5.4. The script defines a *foo* package, which contains a scalar *$x* initialized with the integer 1, and a subroutine *bar* to print the given arguments concatenated with the value of scalar *$x* (i.e., 1). Subroutine *bar* is called from inside the *main* package.

```
package foo;
$x = 1;
sub bar {
   my $args = shift;
   print $args . $x;
}

package main;
foo::bar "Example ";
```

Figure 5.4: A simple Perl script

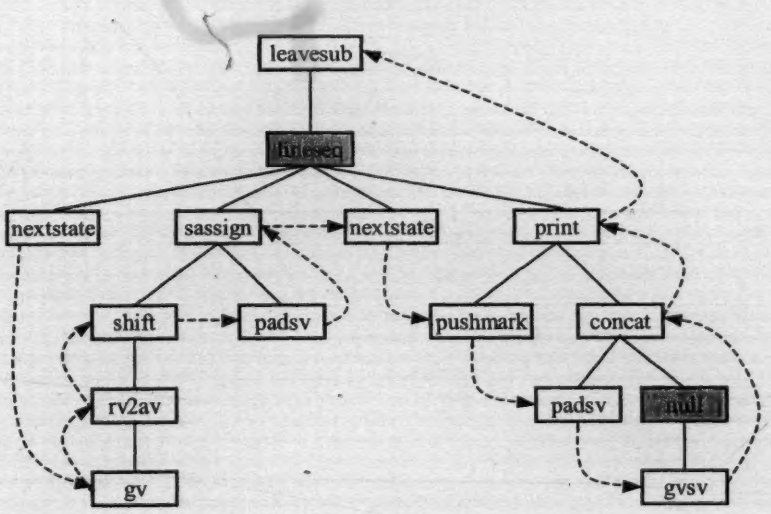

Figure 5.5: Perl operation tree

Figure 5.5 shows the operation tree for the *foo::bar* subroutine, as well as the execution order of the operations. The plain lines represent the containment relation between two operations in the tree. For example the *lineseq* operation has the following four operations as children: *nextstate*, *sassign*, *nextstate*, and *print*. Perl makes several compilation optimization steps, one of them optimizing the operations on constant scalars (e.g., scalar x is always the constant 1). The gray nodes (*lineseq* and *null*) denote operations that are skipped during the execution. The *null* operation was previously an *rv2sv*

operation, which has been optimized away. The dashed lines show the execution flow of the operations (a post-order traversal of the operation tree). The *foo::bar* subroutine starts the execution with the leftmost operation, *nextstate*, then executes a *gv* operation, then executes an *rv2av* operation, and so on.

When Perl executes a script, run-time information is stored in several stacks, of which the most important is the *argument stack*. This stack keeps only the parameters of subroutines, and is used by *ppcode* functions. When a subroutine calls another subroutine, it pushes the actual parameters on the argument stack then makes the call. The called subroutine retrieves the parameters from the stack, executes its operation tree, and pushes the results back on the stack.

5.1.3 Perl Namespaces

Each Perl module has an associated namespace, called a package. Perl assigns a symbol table, called a *stash* (symbol table **hash**), for each package to store entities, such as variables and routines, defined by the package.

By default each Perl program has a *main* package. A single stash is defined for *main*. A package can also define subpackages. In particular, all user-defined, top-level packages are treated as subpackages of *main*. Therefore, a stash may contain links to other stashes. To refer to entities from the stash of a package, a qualified name is used. For example, in the example from Figure 5.4, to execute the *bar* subroutine of the package *foo*, we use the name *foo::bar*.

Perl allows the same name to be used again in a package for different entities. To overcome the problem of duplicates (which cannot exist as keys in a hashtable), Perl creates for each named entity a glob value structure, with entries for the possible types of entities. A glob contains references to objects of the following basic Perl types: scalar (integer, floating-point, or string), array, hash, subroutine, filehandle, and format name. For example, the name *bar* in the package *foo* represents the subroutine *bar*, but it could also have been at the same time initialized to a scalar, an array, or a hashtable (accessed by *$bar*, *@bar*, or *%bar*, respectively).

Figure 5.6 illustrates a simplified version of the internal compile-time representation of the stashes for the Perl script example given in Figure 5.4. (Some fields and chains of links have been omitted.) The figure contains: two stashes (*main* and *main::foo*) for the two packages, five globs (*main::stdin*, *main::foo::bar*, *main::foo::x*, *main::main::*, and *main::foo::*) for three entities and the two stashes, one filehandle structure for *main::stdin*, one subroutine structure for *foo::bar*, and one undefined scalar structure for *foo::x*. *PL_defstash* is a global pointer, which points to the *main* stash. Stashes are internally stored as hashtables, whose elements are pointers to glob structures. The *main* stash contains pointers to the following globs: *main::stdin*, *main::main::*, and *main::foo::*. A glob points to a stash if its name ends in "::",

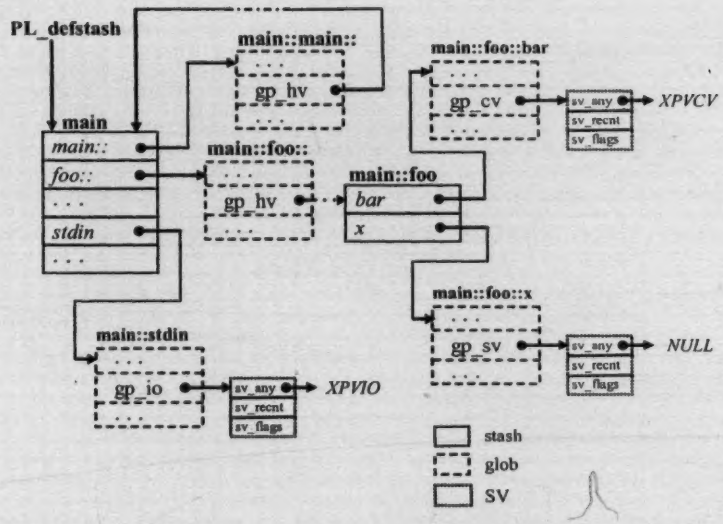

Figure 5.6: Perl internal stashes representation

indicating a namespace. For example, the *main::foo::* glob points to the stash for *main::foo*. The *main::stdin* glob has a pointer to a filehandle structure (an *IO* structure whose *sv_any* field points to an *XPVIO* structure). In the same way, the *main::foo::x* and *main::foo::bar* globs in the *main::foo* stash point to an undefined scalar structure (type determined at run-time) and a subroutine structure, respectively. Traversing the links, we can retrieve the facts from all the stashes. The *main::main::* glob points back to the *main* stash. This ensures that any name combinations of the *main* package are defined (e.g., the stash pointer for *main::main::main* is equal to *PL_defstash*).

5.2 Perl Extractor

This section describes how we "hooked" our component inside the Perl interpreter code to extract facts from a Perl script, largely based on the Perl internal structures explained in Section 5.1. We explain the intepretation process for a Perl script in Section 5.2.1, the instrumentation point of our component in Section 5.2.2, and the extractor implementation in Section 5.2.3.

5.2.1 Interpreting Perl Scripts

An overview of how Perl interprets a script is illustrated in Figure 5.7. The script is parsed and the compiler constructs corresponding operation trees, based on Perl predefined operations. The optimizer component optimizes these trees. After these steps, data structures such as stashes and initial scratchpad structures are populated. Perl is then ready to run the script by traversing the

operation tree, creating further scratchpads, modifying the argument stack, and changing data values. We inserted our component (shaded in gray in Figure 5.7) to output Perl facts just before actually running the script.

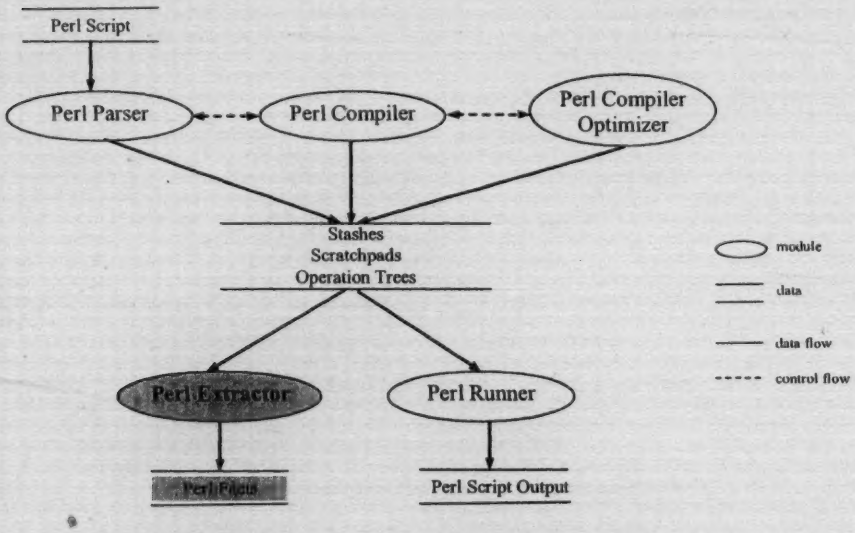

Figure 5.7: Perl interpreter architecture

The Perl parser and compiler are interleaved in the first phase of the interpretation process. The Perl parser uses *yacc*, which works bottom-up, building parse tree(s) from the Perl input script. The Perl compiler builds the operation tree(s) bottom-up driven by yacc. When creating each node in an operation tree, the Perl compiler executes *check routines*, which optimize the current node if necessary. The check routines can add/delete/modify operation nodes above/below in the current operation tree. A check routine has the name prefix *ck_*, and is called from *new*OP* functions. At this point, the operation tree does not have any backlinks for the execution.

If the Perl parser creates a node that provides context for its descendants, then the Perl compiler invokes a top-down pass for the operation subtree for this node, by propagating the context. This builds part of the execution order, but not fully. The execution order is determined when the parser reduces a subroutine or file. Here, the Perl optimizer performs neither a bottom-up or top-down traversal, but an execution order traveral. If the operation tree contains back-references, optimizations cannot remove certain operation nodes. These are converted to null operations, and maintained in the operation tree (see the example operation tree in Figure 5.5). The source code for the Perl parser, compiler and optimizer share the following files: *perly.*{*h,c*}, *toke.c* (the Perl lexer), and *op.*{*h,c*}. Code for the Perl compiler optimizer is found in the *Perl_peep* function (in *op.c*).

5.2.2 Perl Interpreter Control Flow

Here, we focus on the control flow of the Perl interpreter to understand what happens in the life of a script. Figure 5.8 illustrates an overview of the most important functions of the call tree. The file where the function resides is added after the function name in the square brackets. The line representing the call to *Perl_runpops_debug* is dashed because in our Perl extractor this function will not be executed. The place where our extractor is called is in bold. More details about the extraction algorithm is given in Section 5.2.3. The interpreter starts (as does every C program) with *main*. This short function calls only *RunPerl*. *RunPerl* performs three major steps: initializes a Perl interpreter structure; parses, compiles and optimizes the Perl script by calling *perl_parse*; and runs the script by calling *perl_run*.

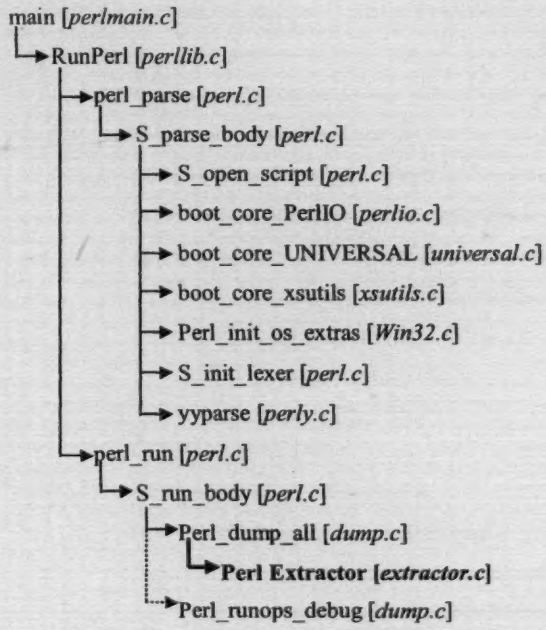

Figure 5.8: Overview of Perl call tree

The function *perl_parse* initializes some global structures, and calls *S_parse_body*. *S_parse_body* opens the script file by calling *S_open_script*, then it initializes some core stashes with external C subroutines needed by almost every Perl script. Next *S_parse_body* calls *boot_core_PerlIO*, *boot_core_UNIVERSAL*, *boot_core_xsutils*, *Perl_init_os_extras*. *S_parse_body* also creates some scalars predefined in every Perl script, such as: *STDIN*, *stdin*, *STDOUT*, *stdout*, *STDERR*, *stderr*, *ARGV*, *ENV*, and *INC*. Then, it calls *S_init_lexer* to initialize some global pointers needed for the lexer and parser. The *yyparse* function is called to parse and compile the script, and to optimize the operation tree.

The *yyparse* function also fills the stashes and the scratchpads associated with each Perl subroutine.

If the function *perl_parse* succeeds, then *RunPerl* calls *perl_run*, which does some initialization, and calls *S_run_body*. If the Perl interpreter is built with the macro *DEBUGGING* defined, then the user can use the Perl debug feature (specified by *-D* options to the Perl interpreter). In this case, Perl keeps all the structures defined in Section 5.1. We also override the flag *-Dx* to invoke our component. Function *S_run_body* calls *Perl_dump_all*, which calls our Perl extractor, instead of dumping the operation trees (what *-Dx* normally does).

We modified the Perl interpreter to not execute the script afterwards; we commented out lines 1926 to 1935 in *perl.c*. These lines call *Perl_runops_debug* if the *DEBUGGING* macro is defined, or *Perl_runops_standard* (in *run.c*) otherwise. These two functions are almost the same, the single difference being that *Perl_runops_debug* dumps various information depending on the debug options provided, while *Perl_runops_standard* does not. Both of these functions traverse the operation trees in execution order, by executing the *ppcode* function associated with each operation node visited. The source code that achieves this process is very short, just three lines of code.

5.2.3 Perl Extractor Implementation

We developed a Perl fact extractor by hooking a new C component to the Perl interpreter. Given a Perl script for analysis, we use the interpreter to build up the corresponding intermediate representation in data structures which are then traversed to obtain facts. The Perl extractor outputs the facts just before executing the script. At this time, most internal structures have been populated: the stashes, the initial scratchpads, and the operation trees associated with the subroutines. Also, the execution order for the operation trees is set in each operation node.

The Perl extractor component contains two files: *schema.h*, which defines the Perl schema used for the output factbase, and *extractor.c*, which contains the implementation of the Perl extractor. The Perl extractor component consists of about 3 KLOC. We modified the *Makefile* by adding our files to be compiled.

An overview of how the Perl extractor works is given in Figure 5.9. The Perl extractor can be divided into five major steps: initialization of its environment, processing the stashes, processing the scratchpads of the subroutines, revealing the cross references, and cleaning its environment.

The Perl extractor starts by initializing its own environment. Several maps are maintained, one map per data type, to assign a unique identifier for each entity. The map keys are the unique complete names of Perl entities, while the values are unique identifiers generated for the keys in the factbase. For a fact from a stash, the key is the name of the stash appended with the name of the entity in the stash (e.g., *foo::bar*). For a lexical variable of a subroutine,

the key is the qualified name of the subroutine appended with the name of the variable (e.g., *foo::bar::args*).

```
INPUT: Perl data structures

initialize Perl extractor environment
  open factbase files
  initialize map ids
  create schema file
  write factbase header

put main stash on stash queue
while stash queue not empty
  get current stash
  output package
  for each entry in current stash
    if entry is a CV
      add entry to subroutine list
      output entry
    else if entry is a stash
      add entry to stash queue
    else
      output entry (for SV,AV,HV,IO,FM)

for each subroutine in subroutine list
  for each lexical in subroutine scratchpad
    output lexical

for each subroutine in subroutine queue
  traverse operation tree in execution order
    output cross references

clean Perl extractor environment
  write factbase footer
  close factbase files
  free map ids

OUTPUT: schema and factbase files
```

Figure 5.9: Perl extractor pseudocode

Next, the Perl extractor reveals global facts by traversing the stashes recursively by starting from the *main* stash (see Figure 5.6). A stash may contain as entries links to other stashes. Each stash is processed by iterating over all the entries in the stash. The entries in a stash (such as Perl arrays, hashtables, subroutines, filehandles, and formats), as well as the name of the stash (a package name) are written to the output files, together with their attribute

information. We build a list of Perl subroutines, and a queue of encountered stashes still to process. The algorithm repeats the process for the next stash retrieved from the queue. This phase ends once the stash queue is empty.

Next, the Perl extractor takes each subroutine from the subroutine list populated by the previous step, and processes its initial scratchpads. This step reveals the Perl lexical variables. Then, the Perl extractor reveals cross references, such as uses of variables or calls to subroutines by traversing the associated operation tree in execution order. The last step closes the factbase files and frees the maps.

5.3 Perl Schema

In this section we describe the Perl schema. The schema is mapped to two different XML-based formats: GXL and Clare formats.

For the GXL format, the output factbase file is *perl-instance.gxl* and the schema file is *perl-schema.gxl*. For the Clare format, the output factbase file is *perl-instance.fb* and the schema file is *perl-schema.xsd*. All the four files are produced at one time.

The Perl schema is illustrated in Figure 5.10. The node types are denoted by boxes. Each node type has its name labeling the box, followed below by its attributes. The Perl schema defines nine node types: *Namespace* (package), *Function* (subroutine), *File*, *GlobalVar*, *LocalVar* (lexical), *PerlIO* (filehandle), *PerlFormat* (format), *Literal*, and *PerlOp* (operation). Each node type has an attribute *name* that denotes the name of the entity. The *Function* node type has in addition the following attributes: *xsub*, whether the subroutine is defined externally in another language [XS]; *sourcefile*, the file where the subroutine is defined; *line*, the starting line number of the subroutine; and *endline*, the ending line number of the subroutine.

There are three possible edge types between the node types: *calls* (dashed line), *uses* (plain line), and *defines* (dotted line). For example, there is a *calls* edge type from *Function* to *Function*, and also from *Function* to *PerlOp*. All three edge types have the attributes *sourcefile* and *line*. In addition to these two attributes, *uses* and *defines* edge types have the Perl *type* attribute to maintain the type (scalar, array, or hashtable) of the destination node.

There are three levels of abstraction for generating our GXL instance factbase. The high level represents the GXL schema that contains elements for representing general graphs such as *node*, *edge*, *rel* (hyperedges), or *attr* (attributes). The middle level represents the Perl schema and is described using GXL. This level defines the particular element types for the Perl language, i.e., the graph in Figure 5.10. Finally, the lowest level uses XML *xlink*s [XML], to describe the "real" facts.

The Clare format, in contrast with the GXL format, focuses more on a specific representation of the facts by skipping the high and part of the middle

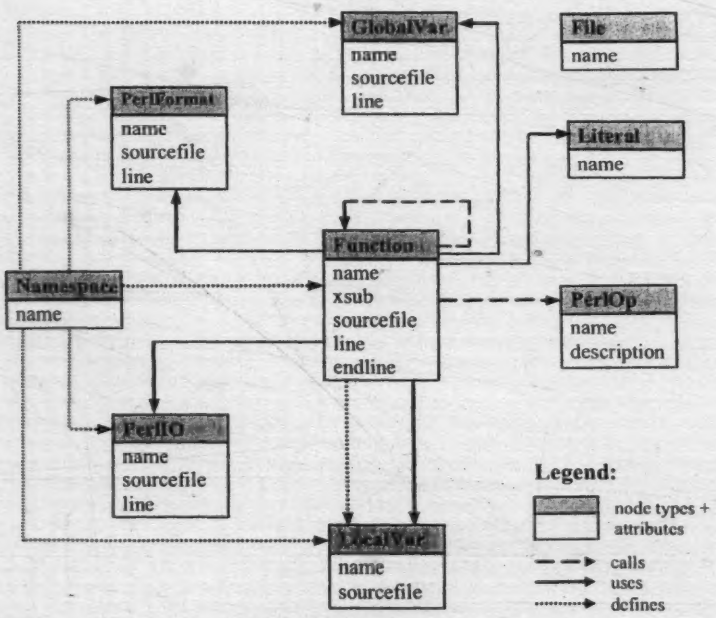

Figure 5.10: Perl schema

levels of the GXL format. The schema of the factbase in Clare format is described in an XSD (XML Schema Definition) file [XSD]. The tags in the instance XML file are the Perl node and edge types described in Figure 5.10. The attributes are represented in XML in the form "*attr_name=attr_value*", where the *attr_name* is an attribute of a node or edge type from the Perl schema, and *attr_value* is the value of the attribute. Each edge has attributes *to* and *from* denoting *id*s of the source and destination node of the edge. The Clare format contains information that is easy to read from both human and machine points of view. Also, representing the factbase in this format is more compact than in the GXL format.

5.4 Perl Results

This section presents and discusses the results and the performance of the Perl extractor. First, we show the output of the extractor for the example from Figure 5.4. Then, we provide the results of a test on all the Perl extension modules, provided by default with the Perl distribution.

5.4.1 Perl Factbase Example

Figure 5.11 illustrates the output of the Perl extractor in Clare format for the script given in Figure 5.4. For brevity, this example factbase does not

contain the stashes loaded by default in the Perl interpreter (i.e., *attributes*, *utf8*, *CORE*, *DynaLoader*, *IO*, *UNIVERSAL*, *XSLoader*). The extractor is run using the command:

perlextractor -Dx example.pl

where *example.pl* contains the Perl script.

To filter some of the Perl packages from the output, the array *PerlPackageOmit* (in *extractor.c*) keeps the names of packages to omit in the resulting factbase. Also, we can define which of the Perl operations are to be written to the factbase. For this, the array *PerlOutputOp* (in *schema.h*) contains a boolean entry for each of the existing Perl operations to determine whether or not to create output.

The resulting factbase contains all the elements from the *example.pl* script. The correspondence between the Perl script elements and the nodes from factbases is straightforward. For example in Figure 5.11, the package *main* corresponds to node *n2*, subroutine *bar* from the package *foo* corresponds to node *n45*, the literal *1* used to initialize the scalar x from package *foo* corresponds to node *n83*, and so on. The factbase contains also the Perl operations *shift* (node *n85*) and *print* (node *n86*). An interesting subroutine is *main::main* (node *n1*), because it does not appear in the Perl script. Any Perl script has by default an entry point represented by the *main::main* subroutine. The global variable *main::_* is predefined, and used by all Perl subroutines to retrieve the input arguments.

5.4.2 Perl Extractor Test

We evaluated our extractor using all the modules provided with the Perl distribution source code. These modules reside in the *ext* subdirectory in the source code, and consists of about 100 source files (46 KLOC).

To extract the facts from all of the modules, we wrote a Perl script that includes all the modules. This Perl script consists of lines of the form *"use <modulename> ();"*, each of which forces Perl to create an intermediate representation for *modulename*. This is how multi-file Perl applications are constructed. The interpreter provides this support automatically, so there is no need to generate separate factbases, do linkage, etc.

The factbase file in GXL format is about 66 MB, while the factbase file in Clare format is only 18 MB in size. We expected that the GXL factbase to be much bigger than the corresponding Clare factbase, as discussed in Section 5.3. The running time spent to the point where our component starts is 12 seconds. The time spent specifically by our component is 40 seconds, for a total running time of 52 seconds for the entire process, which is reasonable for the size of the input. The machine used for evaluation was an Intel Pentium 4 2.6MHz, 1GB RAM, 40GB hard drive at 7200 RPM, running Windows XP. The Perl extractor reveals 517 packages, 8750 subroutines, 2531 global variables, 11352

```
<?xml version="1.0" encoding="ASCII" ?>
<perl xmlns="http://www.perlfactbase.org">
  <Function id="n1" name="main::main" xsub="false" />
  <Namespace id="n2" name="main" />
  <PerlIO id="n4" name="main::stderr" />
  <PerlIO id="n6" name="main::stdout" />
  <PerlIO id="n8" name="main::stdin" />
  <PerlIO id="n9" name="main::ARGV" />
  <GlobalVar id="n10" name="main::ARGV" />
  <GlobalVar id="n11" name="main::INC" />
  <GlobalVar id="n12" name="main::ENV" />
  <PerlIO id="n18" name="main::STDOUT" />
  <GlobalVar id="n20" name="main::_" />
  <PerlIO id="n22" name="main::STDERR" />
  <PerlIO id="n23" name="main::STDIN" />
  <Namespace id="n44" name="main::foo" />
  <Function id="n45" name="foo::bar" xsub="false"
     sourcefile="example.pl" line="3" />
  <GlobalVar id="n46" name="foo::x" sourcefile="example.pl"
     line="2" />
  <LocalVar id="n82" name="foo::bar::$args"
     sourcefile="example.pl" />
  <Literal id="n83" name="1" />
  <uses from="n1" to="n83" type="int" sourcefile="example.pl"
     line="2" />
  <uses from="n1" to="n46" sourcefile="example.pl" line="2" />
  <Literal id="n84" name="Example " />
  <uses from="n1" to="n84" type="string" sourcefile="example.pl"
     line="9" />
  <calls from="n1" to="n45" sourcefile="example.pl" line="9" />
  <PerlOp id="n85" name="shift" />
  <calls from="n45" to="n85" sourcefile="example.pl" line="4" />
  <defines from="n45" to="n82" type="scalar" sourcefile="example.pl"
     line="4"/>
  <uses from="n45" to="n82" type="scalar" sourcefile="example.pl"
     line="5" />
  <uses from="n45" to="n46" sourcefile="example.pl" line="5" />
  <PerlOp id="n86" name="print" />
  <calls from="n45" to="n86" sourcefile="example.pl" line="5" />
</perl>
```

Figure 5.11: Factbase for the Perl example in Clare representation

lexical variables, 83 filehandles, 8477 literals, and 126585 relations among these nodes.

5.5 Summary

This chapter presents an approach to create a fact extractor for Perl code by inserting a component into the Perl interpreter itself. The Perl extractor is part of the fact extraction component in our architecture. This extraction component consults Perl internal data structures, which are populated accordingly for the input script by the interpreter. These data structures, the interpretation process, the component insertion point, the extraction algorithm, and output representation are described in detail.

In general, we believe a similar approach could extract facts for another language implemented by an interpreter. A chief advantage of the described approach is reusing an authoritative implementation of the language's behavior, rather than developing a fact extractor from scratch. The Perl interpreter provides much support, such as dealing with globs, namespaces, implicit global variables like _, and multi-file applications. One limitation is that the deep dependence on the internal data structures makes the extractor sensitive to interpreter implementation changes. Still, we address this issue by not referring to the fields of structures directly but use the provided macros. Another limitation is that the static analysis is lightweight, so dynamic behavior as found in object-oriented Perl code is not fully considered.

Chapter 6

Analyzing and Representing Cross-Language Dependencies

The previous two chapters presented finding the entities and relationships only inside one language at a time (for C/C++, Java, Perl). This chapter enhances the set of existing facts with new ones representing the cross-dependencies among code in different languages. Section 6.1 presents how the dependencies between Java and C/C++ can be retrieved. Following, Section 6.2 describes the cross dependencies between from Perl to C. Section 6.3 describes the cross dependencies between from Tcl to C. Section 6.4 describes the cross dependencies between from Python to C. The commonalities among the extension mechanisms of scripting languages is summarized in Section 6.5. Finally, Section 6.6 presents a common schema used to represent the same type of facts among different languages.

6.1 Recognizing Java to/from C/C++

In this section we discuss the Java Native Interface (JNI) [JNI] mechanism that allows to connect Java and C/C++ languages. We present also the algorithm that finds these relationships between Java and C/C++.

In C++, JNI functions are defined as inline member functions that expand to their C counterparts. The underlying mechanism is exactly the same as using C. Therefore, we will resume to discuss the JNI mechanism using C.

Calling C from Java

Java calls C or C++ functions using JNI. With Java and JNI, one writes a Java program that dynamically loads a native library that contains implementations of one or more native methods declared in the Java class. We discuss the case when the dynamic link library is created using C. Generally, there are six steps to follow to create such a connection between code written in Java and C/C++:

1. Create a Java source file that declares a class with one or more native

methods.

2. Compile the Java source file to create a .class file.

3. Use the *javah* generator with the *-jni* switch to automatically create a header file for use in the C program.

4. Create a C source file that implements the native method(s).

5. Compile the C source file to create a dynamic link library that exports the native method(s).

6. Run the Java program using the Java virtual machine.

JNI uses a standard naming and calling convention, so that the Java Virtual Machine (JVM) can recognize and invoke the C functions corresponding to the Java native methods. We provide an example to understand better the dependencies between Java and C using JNI. Suppose that we have the following *Test* Java class:

```
public class Test {
  static { System.loadLibrary("Test"); }

  public int iValue;
  public Test() { }
  public double compute(Vector v, float f)
  { . . . }
  public native void print(String msg);

  public static void main(String[] args) {
    Test t = new Test();
    t.print("Hello from C!");
  }
}
```

Figure 6.1: *Test* Java class

The *Test* class contains a *print* native method and a *main* method that invokes the *print* method. The constructor, *iValue* instance variable, and *compute* method of the *Test* class will be used later to illustrate the access of Java from C. Also, the class loads dynamically a library that provides the implementation of the *print* method in C. Executing the *javah* generator with the *-jni* option for the *Test* Java class, we obtain a *Test.h* file that contains a C function declaration corresponding to the *print* native method:

```
void JNICALL Java_Test_print(JNIEnv *, jobject, jstring);
```

A cross-language dependency exists between the *print* Java native method and the *Java_Test_print* C method. Note that the name of the corresponding Java native method consists of *'Java'*, *'_'*, followed by the name of the Java class for the native method (e.g., *Test*), *'_'*, and the name of the Java native method (e.g., *print*). Also, the signature of the corresponding C function consists of three argument types. The first two are required by the JNI framework to access its functions. The third argument type is *jstring*, which is the corresponding C type for the Java argument type *String* for the *print* native method.

To match up signatures of Java native methods and C functions correctly, we need to know the mapping between the Java types and C types defined by JNI (see Table 6.1). For the primitive Java types the corresponding JNI types defined in C have the same name preceded by *'j'*. The JNI C type for a Java array of primitive types is: *'j'*, the name of the Java primitive type, followed by *'Array'*. For example, for the Java array *int[]*, the JNI C type is *jintArray*. There are also some JNI C types defined for Java complex types.

Java Type	**JNI C Type**
boolean	jboolean
byte	jbyte
char	jchar
short	jshort
int	jint
long	jlong
float	jfloat
double	jdouble
void	void
java.lang.Object	jobject
java.lang.String	jstring
java.lang.Class	jclass
java.lang.Throwable	jthrowable
java.lang.Object[]	jobjectArray
Type derived from Object	jobject
Array of Type derived from Object	jobjectArray

Table 6.1: Mapping Java and C/C++ Types

JNI uses the conventions presented before to handle the name-mangling for native Java methods that are not overloaded. However, JNI supports to overload the native Java methods. In this case, JNI adds in addition to the name-mangling conventions for non-overloaded native Java methods another component. This component adds two underscores ("__") followed by the mangled argument signature to the native method name presented in Table 6.2. This case of JNI name-mangling is applicable only for the overloaded native

Java methods. The non-native Java method does not reside in the native library. In the following example, JNI uses the name-mangling conventions for non-overloaded native Java methods to resolve the native method *f*.

```
void f();
native void f(int i);
```

Figure 6.2 presents the algorithm for finding the cross-dependencies from Java to C/C++:

```
INPUT: Facts from Java and C/C++
1. for each Java class c
2.   for each native method m in c
3.     map arg types of m to JNI C types
4.     find C function Java_c_m with the corresponding
           JNI C argument types
5.     dependency: m calls Java_c_m
OUTPUT: Dependencies from Java to C/C++
```

Figure 6.2: Algorithm for finding cross-dependencies from Java to C/C++

The input of the algorithm consists of the facts for the Java and C/C++ languages individually. This information is provided after running the extractors for each language (these extractors were described in Chapter 4). The algorithm tries to find a match for every native method (line 2) of every Java class (line 1) in the set of C/C++ functions (line 4). The matching of names and signatures is based on the JNI standard. Finally, the algorithm outputs these dependencies as simple *calls* from Java methods to C/C++ functions.

Accessing Java from C

Conversely, a C function can create, update, and access Java objects. There are two cases regarding how Java may be accessed from the C side. The first case appears when a Java method is implemented as a C function, and the C function calls back on Java objects. The second case occurs when embedding a Java Virtual Machine (JVM) inside a C application.

In both cases, support is provided by the JNI API. JNI provides a set of functions and mechanisms for communicating between Java and C code. That is, JNI is a bridge between the two aforementioned languages [JNI].

To understand better how C functions can access Java classes, methods, or fields, consider a simple example following from the *Test* Java class of Figure 6.1. We modify the *Java_Test_print* C function that implements the Java native method *print* to access the *iValue* and *compute* members of the *Test* Java class. The C/C++ code is listed in Figure 6.3.

The Java class is retrieved in C using the *GetObjectClass* JNI function. To access a Java method in C, we use *GetMethodID* with the name of the Java method and its encoded signature (see Table 6.2). To find the desired Java

```
JNIEXPORT void JNICALL Java_Test_print(JNIEnv *env, jobject obj,
                                       jstring s)
{
  jclass cls = (*env)->GetObjectClass(env, obj);
  jmethodID mid = (*env)->GetMethodID(env, cls, "compute",
                                      "(Ljava/util/Vector;F)D");
  . . .
  (*env)->CallDoubleMethod(env, obj, mid, v, 1.0f);
  jfieldID fid = (*env)->GetFieldID(env, cls, "iValue", "I");
  int i = (*env)->GetIntField(env, obj, fid);
  . . .
}
```

Figure 6.3: C sample code for accessing Java from C code

method, JNI searches in the symbol table based on the given method name and its encoded signature. We invoke the Java method using a *Call*<T>*Method* JNI call, where T denotes the type returned by the method (e.g., if the method returns an *int*, then T will be *Int*). The *iValue* Java field is accessed using *GetFieldID* and respectively the *GetIntField* JNI function.

JNI provides an encoding scheme for Java method signatures and field types. Table 6.2 illustrates the possible encodings defined by JNI. For example, the *compute* Java method has two arguments of type *java.util.Vector* and type *float*, and returns a *double*. The encoded signature of *compute* is therefore *(Ljava/util/Vector;F)D*. The '.' character used for packages in Java, is replaced by the '/' character.

Java Type	**JNI Encoding**
boolean	Z
byte	B
char	C
short	S
int	I
long	J
float	F
double	D
Java_class	L*Java_class*;
type[]	[*type*
method signature	(*arg_types*)*ret_type*

Table 6.2: Encoding Java Signatures

We developed an algorithm to find the dependencies from C to Java. Because our approach uses a simple static analysis, some heuristics are used to determine those points in C files where there is an access to JNI and find the Java classes, methods and fields specified as static encoded strings in C. Deter-

mining all these dependencies is difficult because of the potential for dynamic string manipulation.

```
INPUT: Facts from Java and C/C++
 1. for each c call from caller f to callee g
 2.   set P array to parameters of g
 3.   if type of P[0] is ('JNIEnv*' or 'JavaVM*')
 4.     if g is ('Call<T>Method' or 'CallStatic<T>Method')
 5.       retrieve Java class C using P[1]
 6.       retrieve Java method m using P[2]
 7.       if m is method of C
 8.         dependency: f calls m
 9.     if g is ('Get<T>Field' or 'GetStatic<T>Field' or
                 'Set<T>Field' or 'SetStatic<T>Field')
10.       retrieve Java class C using P[1]
11.       retrieve Java field x using P[2]
12.       if x is field of C
13.         dependency: f uses x
OUTPUT: Dependencies from C/C++ to Java
```

Figure 6.4: Algorithm for finding cross-dependencies from C/C++ to Java

Figure 6.4 presents the algorithm for finding the cross-dependencies from C/C++ to Java. The algorithm takes as input the facts from Java and C. Line 1 inspects each call from the C side. Line 3 checks and report if it is a call using JNI. Then, the algorithm checks if the callee is one of the JNI API functions (i.e., *Call<T>Method*, *CallStatic<T>Method*, *Get<T>Field*, *Set<T>Field*, *GetStatic<T>Field*, or *SetStatic<T>Field*, where *<T>* can have of the following values: "Void", "Boolean", "Char", "Byte", "Short", "Int", "Float", "Double", or "Object") for accessing Java methods (line 4) or fields (line 9). For each specific JNI function, the algorithm finds the possible Java method (lines 5-8), or field (lines 10-13) used by the call. The Java class is retrieved from the arguments of the JNI functions *GetObjectClass* or *FindClass* (lines 5,10). The Java method is retrieved from the arguments of the JNI functions *GetMethodID* or *GetStaticMethodID* (line 6), while the Java field is retrieved from the arguments of the JNI functions *GetFieldID* or *GetStaticFieldID* (line 11).

6.2 Connecting Perl to C

Calling C functions from Perl can have several advantages, such as improving the speed of a Perl script by rewriting the time-consuming routines in C, accessing low-level system calls and libraries, or accessing legacy applications that expose a C interface. For example, the Perl B module (module provided by default in the Perl distribution) uses this interoperability mechanism. Consequently, we need to recognize the appearance of such code in a mixed Perl

and C system.

To call a C function from Perl, developers need to write the necessary glue code for the Perl interpreter. The glue code usually contains two files: a module file in Perl with the *.pm* extension, and a C file. The Perl module tells the Perl interpreter how to load, dynamically or statically, the library that contains the C function, and the C file puts the C function in the context of the Perl interpreter and associates a new Perl routine with the C function. When we call this Perl routine from a script, the C function associated with the Perl routine is executed. In Perl's terminology, such a C function is also known as an external subroutine or *XSUB* function.

To understand this process better, consider a simple example. Suppose that we want to create a Perl module *Test* that contains a routine called *test*, and that we want to implement this *test* routine as a C function, instead of a plain Perl routine. The *test.c* file listed in Figure 6.5 illustrates how the C function (*XS_Test_test*) can be implemented, and how the C function can be registered to the Perl interpreter via *boot_Test*.

```
1. #include "perl.h"
2. #include "XSUB.h"
3.
4. XS(XS_Test_test);
5. XS(XS_Test_test) {
6.   dXSARGS;
7.   if (items != 0)
8.     Perl_croak(aTHX_ "Usage: test()");
9.
10.  printf("Test: Perl calls C!\n");
11.
12.  XSRETURN_EMPTY;
13. }
14. XS(boot_Test);
15. XS(boot_Test) {
16.  dXSARGS;
17.  char* file = __FILE__;
18.
19.  XS_VERSION_BOOTCHECK ;
20.  newXS("Test::test", XS_Test_test, file);
21.  XSRETURN_YES;
22. }
```

Figure 6.5: Listing of *Test.c* file for Perl

The *perl.h* header file declares C functions that access the Perl internal data structures, and the *XSUB.h* header file defines a set of macros to write Perl external subroutines. In this example, the C function *XS_Test_test* is the C portion of the glue code. In practice, it is a kind of wrapper which typically delegates other C functions to do the real work.

Three macros from *XSUB.h* hide much of the details on how the C function and Perl internals interact:

- **XS** – The *XS* macro defines the standard signature for a new XSUB:

  ```
  #define XS(name) void name(PerlInterpreter *pi, CV *cv)
  ```

  Note that an XSUB function takes two parameters and returns nothing. The first parameter *pi* is a pointer to the current Perl interpreter. The second parameter *cv* is a Perl data structure that represents this function inside the Perl runtime. Other function-specific arguments are made available to an XSUB function implicitly through the Perl runtime stack. Before an XSUB function is invoked, the actual parameters are pushed onto the Perl runtime stack, which can be accessed by the XSUB function through the set of macros defined in *XSUB.h*.

- **dXSARGS** – The *dXSARGS* macro defines the necessary variables for manipulating the Perl stack. For instance, the variable *items* in the example is an integer containing the number of arguments pushed onto the stack by the caller.

- **XSRETURN_EMPTY** – The macro *XSRETURN_EMPTY* indicates that this subroutine does not put anything on the stack as a return value.

The rest of *XS_Test_test* is explained as follows. Line 7 checks if this function has any parameters, and if it does, a usage message is then printed, and the function returns. Note that *Perl_croak* is a Perl internal function which takes two parameters. The first parameter *aTHX_* is a macro that defines a pointer to the current Perl interpreter followed by a comma. The second parameter is the string to be displayed. Line 10 prints a simple message to standard output.

In practice, a Perl module may have a number of such C functions to be called. These C functions comprise a module-specific extension to Perl, and are made known to the interpreter using a specially named registration C function. When a command is issued for Perl to load the module, this registration function is called first. In Figure 6.6, the function *boot_Test* is the registration function for the Perl module *Test*. This function makes the *Test::test* Perl routine known to the Perl interpreter, and associates the C function *XS_Test_test* with the Perl routine *Test::test*. In general, the name of the registration function is formed by prefixing the module name with *boot_*.

The *XS_VERSION_BOOTCHECK* macro checks the module version. The *newXS* macro associates the *XS_Test_test* C function with a Perl subroutine called *test* in a module called *Test* (*Test::test*).

Perl provides two mechanisms for integrating C extensions. One way is to statically link the extension into the Perl interpreter code itself. The other way

is to dynamically load the extension as a C library. Figure 6.6 lists the *Test.pm* file that helps to dynamically load the *Test* module. The package statement at line 1 introduces the namespace associated with the module. The package name must match the module name.

```
1. package Test;
2. use strict;
3. use warning;
4.
5. our $version = '1.0';
6. require DynaLoader;
7. bootstrap Test $version;
8. ...
```

Figure 6.6: Listing of *Test.pm* Perl module file

We can load this module using *use Test;*, and call the Perl routine using *Test::test;*. When the Perl interpreter sees a *use Test;* statement, it searches for a *Test.pm* file to load in all the paths in the predefined *@INC* array. The required *DynaLoader* module is a predefined Perl module that loads shared libraries at runtime (line 6). Perl bootstraps the *Test* module for the given version (line 7). Here, Perl calls its dynamic loader routine, loads the shared library built from the *Test.c* file, and executes the *boot_Test* C function to initialize the *Test* module with the subroutines defined by *newXS*.

6.3 Connecting Tcl to C

The Tcl scripting language allows adding new functionality implemented in C. To call a C function from Tcl, developers need to write the glue code necessary to define the function and to register the new command with the Tcl interpreter.

We illustrate the process with a simple example that creates two Tcl commands, called *stest* and *otest*. These commands are implemented as two C functions. The *test.c* file listed in Figure 6.7 contains the corresponding C functions (*stest* and *otest*), and the function (*Test_Init*), which registers the C functions with the Tcl interpreter.

The header file *tcl.h* contains the interfaces for accessing the Tcl internals. Two different ways are shown of defining and registering commands to the Tcl interpreter. The C function to be called from Tcl must use one of two signatures, as demonstrated by *stest* at lines 3 and 4, and *otest* at lines 9 and 10. The signature of *stest* takes four arguments: client data, the interpreter in which the command is executed, the number of string parameters, and the array of string parameters passed.

Function *otest* also has four parameters, with the first two the same as in the signature of *stest*. The last two parameters are the number of parameter

```
1. #include "tcl.h"
2.
3. int stest(ClientData cd, Tcl_Interp *ti,
4.       int argc, char *argv[])
5. {
6.   printf("Test: Tcl calls C!\n");
7.   return TCL_OK;
8. }
9. int otest(ClientData cd, Tcl_Interp *ti,
10.         int objc, Tcl_Obj *CONST objv[])
11. {
12.   printf("Test: Tcl calls C!\n");
13.   return TCL_OK;
14. }
15. int Test_Init(Tcl_Interp *ti){
16.   Tcl_CreateCommand(ti, "stest",stest,
17.         (ClientData)NULL,
18.         (Tcl_CmdDeleteProc*)NULL);
19.   Tcl_CreateObjCommand(ti, "otest",
20.          otest, (ClientData)NULL,
21.         (Tcl_CmdDeleteProc*)NULL);
22.   Tcl_PkgProvide(ti, "Test", "1.0");
23.   return TCL_OK;
24. }
```

Figure 6.7: Listing of *Test.c* file for Tcl

objects and the array of parameter objects passed to this command. Note that in the first case the arguments to the new command are C strings, while in the second case the arguments are Tcl objects. Tcl commands with the second signature have better type-checking support, and may run slightly faster. In the first case, C string arguments have to be converted to Tcl objects. Thus in practice, the second case is recommended for creating a new Tcl command, and the first case is supported only for backward compatibility.

Both *stest* and *otest* print a simple message. They also must return a pre-defined integer value to indicate to the interpreter if an error has occurred during the execution of the command. In our example *TCL_OK* is returned to indicate that there are no errors.

A special C function must be defined to register C functions to Tcl. New Tcl commands may belong to a Tcl package (in our example, the Tcl package is *Test*). The name of this registration function must contain the name of the package followed by *_Init*, and the single parameter of this function must be a Tcl interpreter. In the example, *Test_init* registers the two new Tcl commands.

Corresponding to the two signatures of C functions for Tcl commands, there are two ways of registering a new Tcl command to the Tcl interpreter, using *Tcl_CreateCommand* (for C string arguments) and *Tcl_CreateObjCommand* (for Tcl object arguments). These two functions take the same parameters: the interpreter in which the command is executed, the name of the new Tcl command, the pointer to the associated C function that implements the new command, client data that is passed when executing the new command, and a pointer to a function to be called when the new command is removed from the interpreter.

The *Tcl_PackageProvide* call at Line 22 declares that a Tcl package named *Test* with version *1.0* is made available to Tcl.

To make the extension available, one needs to compile the *Test.c* file and build a new C library. To use the new Tcl commands, the library must be loaded using either of the Tcl commands *load* or *package require*.

6.4 Connecting Python to C

The Python scripting language allows adding new functionality implemented in C. The mechanism for building new Python modules written in C follows almost the same mechanisms as for Perl and Tcl. A C function to be integrated with Python is written, and this function is registered to the Python interpreter. A library is built for the C code, and it is loaded using an existing loading method provided by Python.

A simple example illustrates this mechanism. We create a module called *Test* that contains a Python function *test* implemented in C. The new *test* Python function prints a constant string. The *Test.c* file listed in Figure 6.8 contains the C implementation of the *test* Python function, and the initialization of the *Test* module with the new function.

```
1. #include "Python.h"
2.
3. static PyObject*
4. test(PyObject *self, PyObject *args) {
5.   printf("Test: Python calls C!\n");
6.   Py_INCREF(Py_None);
7.   return Py_None;
8. }
9. static PyMethodDef TestMethods[] = {
10.  {"test", test, METH_VARARGS, "comment"},
11.  {NULL, NULL, 0,NULL} /*sentinel*/
12. };
13. PyMODINIT_FUNC initTest(){
14.   Py_InitModule("Test", TestMethods);
15. }
```

Figure 6.8: Listing of *Test.c* file for Python

The header file *Python.h* contains the interfaces to access the Python internals. To be registered as a Python command, a C function must possess a predefined signature with two parameters. If the function is meant to be invoked on an object, then the first parameter *self* will be a pointer to the receiver Python object, otherwise it will be *NULL*. The second parameter *args* contains the arguments passed to the function.

A Python C function should always return a non-NULL reference to a PyObject. The Python interpreter treats it as an error if a Python C function returns *NULL*. To express the semantics of returning nothing, a function may return a special Python object *Py_None*. However, before returning *Py_None*, the function must increase the reference counter of *Py_None* by calling the *Py_INCREF* macro so that *Py_None* is not to be garbage-collected.

The Python type *PyMethodDef* contains an entry definition, a tuple of four elements that define a Python command. It contains the name of the Python command in the module, the C function that implements the functionality of the new Python command, how to pass the arguments, and a C string comment for the new Python command.

Lines 9–12 defines an array of entry definitions, each of which declares a C function that comprise the *Test* module. In the example, there is only one entry, which associates the Python *test* command with the C function *test*. We use *METH_VARARGS* for passing the Python parameters, which means that Python parameters are passed as a tuple. This is similar to variable length argument lists in C. The tuple can be parsed using the Python API function *PyArg_ParseTuple*.

The function *initTest* creates and initializes the *Test* Python module. This is a special function that informs the Python interpreter of the content of this module. The name of this registration function must contain *init* followed by the name of the module. *Py_InitModule* is a Python function that associates

the new Python module *Test* with the array of entry definitions above.

The *Test.c* file is compiled and a new library is built. To make the new Python module available to the Python interpreter, the library is loaded using the *imp.load_dynamic* Python command, which loads a dynamic library containing a Python module (*imp* is a pre-defined Python module and *load_dynamic* is a method of this module). *imp.load_dynamic* searches in the dynamic library for an entry with a name that starts with *init* followed by the name of the Python module (e.g., *Test*), and executes the corresponding C function. This new module can be imported and used by other Python modules using *import Test*.

6.5 Extension Mechanism Commonalities

The extension mechanisms for scripting languages (such as Perl, Tcl, and Python) are similar, mostly due to the interpreter implementations being written in C. Thus, the basic technique to identify uses of these mechanisms can be generalized with a small language-specific portion. Table 6.3 summarizes the extension mechanisms, with the slight differences for these languages in four source-level respects. Considered are: the necessary header files to include, the right signatures of C function declarations to use, the registration interface to make C functions known to the interpreter, and the loading commands as scripting language code to enable the extension. Because the extension mechanisms are so similar, code generators like SWIG [SWIG] can assist in generating the many different scripting language dependent wrappers associated with a single C function.

	Perl	**Tcl**	**Python**
Header files	*perl.h* and *XSUB.h*	*tcl.h*	*Python.h*
Declaration	*void (PerlInterpreter *, CV *)*	*int (ClientData, Tcl_Interp*, int, char*[])* *int (ClientData, Tcl_Interp*, int, Tcl_Obj*[])'*	*PyObject* (PyObject*, PyObject*)*
Registration	*XS(boot_*packageName*)* *newXS* macro *newXSproto* macro	packageName*_Init* *Tcl_CreateCommand* *Tcl_CreateObjCommand*	*init*ClassName *Py_InitModule* *PyMethodDef* array
Loading	*require DynaLoader;* *bootstrap* Test version	*load* Test *package require* Test	*imp.load_dynamic* Test

Table 6.3: Summary of Perl, Tcl and Python to C extension mechanisms

The cross-language dependencies can be identified from the C facts alone (assuming an accurate C fact extractor). For instance in Perl, one can search for occurrences of the *newXS* macro and associate the new Perl subroutine (e.g., *Test::test*) in the first parameter with the C function in the second parameter (e.g., *XS_Test_test*). In Tcl, one can search for occurrences of the

Tcl_CreateCommand and *Tcl_CreateObjCommand*, and associate the new Tcl command (e.g., *Test::stest*) in the second parameter with the C function in the third parameter (e.g., *stest*). In Python, one can search for occurrences of the *Py_InitModule*, and associate the new Python command (e.g., *Test::test*) in the values of the array of the second parameter with the C function retrieved from values of the same array (e.g., *test*).

Nevertheless, we embarked on a more syntactic and thorough approach to support the exploration of facts from all involved languages (not just C), and to build an infrastructure that would allow for more complete analyses in the future.

6.6 Common Schema

Facts to be extracted from each language could be modeled using separate schemas, but this would lead to many similar entities for common notions like namespaces, subroutines, variables, and calls. To address this problem, we have evolved a simpler, common schema that unifies similar notions across the languages of interest [Moise05]. The fact extractors are designed to produce facts that conform to the common schema. This approach also simplifies the implementation of downstream tools, such as visualizers, that use or present the facts.

Entity Type	C/C++	Java	Perl	Tcl	Python
Class	•	•		○	○
Enum	•	○			
EnumValue	•	○			
File	•	•	•	○	○
Function	•		•	○	○
FunctionDecl	•				
GlobalVar	•		•	○	○
Literal	•	○	•	○	○
LocalVar	•	•	•	○	○
Macro	•				
MemberVar	•	•		○	○
Method	•	•	○	○	○
MethodDecl	•				
Namespace	○	•	•	○	○
Typedef	•				
Union	•				

Table 6.4: Common schema entity types

Practically, the common schema is a union of the schemas presented in Section 4.4 and Section 5.3. In addition, the common schema includes the

schemas for Tcl and Python languages.

Table 6.4 lists the entities in the common schema, and indicates whether each entity is relevant for C/C++, Java, Perl, Tcl, and Python. The first column consists of all the node types of our schema, and the next columns indicate if the node type is available in the language from the first row of the table. For example, we have the *LocalVar* node type exists in C/C++, Java, Perl, Tcl, and Python. An empty cell denotes the node type does not have a corresponding construct in the corresponding language. For example, the *Union* node type does not exist for the Perl, Tcl, or Python languages. An empty dot represents an entity that is not revealed by our extractors.

The relation types between the node types are given in Table 6.5.

	calls	friend of	has	includes	inherits from	uses
Class		Class	MemberVar Method MethodDecl		Class	Class Union
Enum			EnumValue			
EnumValue						
File				File		
Function	Function FunctionDecl Method MethodDecl	Class				Class Enum EnumValue GlobalVar LocalVar Macro MemberVar Union
FunctionDecl						
GlobalVar						
LocalVar						
Macro						
MemberVar						
Method	Function FunctionDecl Method MethodDecl					Class Enum EnumValue GlobalVar LocalVar Macro MemberVar Union
MethodDecl						
Typedef						
Union			MemberVar			Union

Table 6.5: Common schema relation types

The *calls* edge type covers the control dependencies among the subroutine-like entities from different languages. The *uses* edge type covers the control dependencies between a subroutine and a member variable entities from different languages. The *mechanism* attribute for the edge contains more information about the invocation used. For an invocation between Java and C/C++, the *mechanism* attribute has the string value *JNI*. The facts involved in a cross-

dependency come from different factbases. Therefore, two attributes, *fbfrom* and *fbto*, are used to find the factbase of the facts involved in a cross-language dependency.

The attribute types for both node and edge types in the common schema are illustrated in Table 6.6.

Node & Relation Types	Attribute Types
Class	name, sourcefile, line, column, endline, endcolumn, kind (*structdef*)
Enum	name, sourcefile, line, column, endline, endcolumn
EnumValue	name, sourcefile, line, column, endline, endcolumn
File	name, sourcefile, language
Function	name, sourcefile, return_type, arg_names, arg_types, line, column, endline, endcolumn, kind (*inline*)
FunctionDecl	name, return_type, arg_names, arg_types
GlobalVar	name, sourcefile, real_type, line, column, endline, endcolumn
Literal	name, sourcefile
LocalVar	name, sourcefile, real_type, line, column, endline, endcolumn
Macro	name, sourcefile, line, column, endline, endcolumn
MemberVar	name, sourcefile, real_type, line, column, endline, endcolumn, kind (*private*, *protected*, *public*)
Method	name, sourcefile, return_type, arg_names, arg_types, line, column, endline, endcolumn, kind (*private*, *protected*, *public*, *virtual*, *inline*, *constructor*, *destructor*, *purevirtual*)
MethodDecl	name, return_type, arg_names, arg_types
Namespace	name, sourcefile, line
Typedef	name, sourcefile, real_type, line, column, endline, endcolumn
Union	name, sourcefile, line, column, endline, endcolumn
calls	sourcefile, line, column, callee_args, fbfrom, fbto
friend_of	sourcefile, line, column
has	sourcefile, line, column
includes	sourcefile, line, column
inherits_from	sourcefile, line, column
use	sourcefile, line, column, fbfrom, fbto

Table 6.6: Common Schema - Attribute Types

6.7 Summary

This chapter reports our work on extracting cross-language dependencies between Java and C/C++. The resulting factbase conforms to a common schema.

The chapter also presents the mechanisms for adding new commands written in C to three widespread scripting languages: Perl [PERL], Tcl [TCL], and Python [PYTHON]. We use the Perl extension mechanism as the primary example. The core commands of a scripting language can be extended by writing new commands using either the scripting language itself or a system language such as C. There are two main reasons for writing the new commands in C. First, a new command implemented in C is more efficient than the equivalent

implemented in the scripting language. Second, and more importantly, for some tasks, it may not be possible to implement the new commands within the scripting language (e.g., accessing a new low-level system device).

Chapter 7

Evaluation and Applications

This section describes fours tests and applications used to evaluate our approach. For the first test, we investigated the Perl B module which is written in Perl and C. Another small application, called *Win32RegKey* [CoreJava2], written using Java and C is used as a second test. To check its scalability and usefulness, we tested our approach on the *Java-GNOME* system [JavaGNome], which is written also in Java and C. Another case study evaluates our approach for all standard Perl modules, which come with the distribution of Perl interpreter. The languages presented in this test are Perl and C.

7.1 Perl B Module Test

The Perl B module is used as an example to illustrate the exploration of calls in Perl and C code. This module implements the backend of the Perl compiler, which can be used to create opcodes for interpretation. The B module accesses Perl internal data structures through a set of functions defined by the XS mechanism. Other backend utilities, such as cross-reference reports, can be implemented in Perl code, on top of B commands.

Initially, the user specifies the factbases that comprise the system. Figure 7.1 presents the visualization plug-in, after the Perl and C factbases are loaded. The layout loosely follows Eclipse workbench conventions. From left to right, it contains a navigation view, an editor view for cross-language dependencies (Perl subroutines on-the-left associated to C functions on-the-right), backward and forward call graph columnar views, and an outline view of functions organized by language. Colors are used consistently in the views to distinguish the artifacts of differing languages.

Cross-language dependencies can be used for easing the exploration of control flow from one language to another. In the *Calls From* columnar view, a developer can see what functions call a given one function, and follow the calls deeper by exploring more columns to the right. Similarly, the *Calls To* columnar view shows what functions a given one function calls.

A developer may want to know how a Perl external routine is imple-

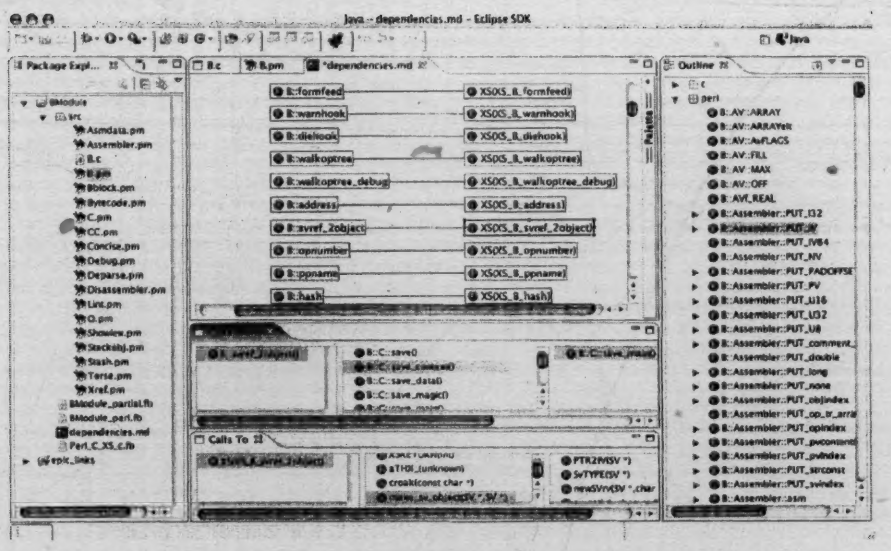

Figure 7.1: Support Perl-to-C dependencies in Eclipse

mented in terms of C functions. For example, as shown in the *Calls To* view, *B::svref_2object* is implemented by a C function *XS_B_svref_2object*, which in turn calls a number of other C functions.

A developer may also want to know what Perl subroutines could be impacted by a change to a C function. For example, considering *Calls From* view, changing *XS_B_svref_2object* may influence other Perl routines that use it directly or indirectly, such as *B::C::save_context* and *B::C::save_main*.

7.2 Win32RegKey Test

The *Win32RegKey* application is provided as an example in [CoreJava2], to illustrate how to call C functions from Java, and how to use Java from C. *Win32RegKey* application accesses the Windows registry from Java using a C-based API. Because accessing the Windows registry is a non-portable feature (being Windows-specific), and Java has no support for accessing the registry, it makes sense to use *Win32RegKey* to gain the needed access. *Win32RegKey* uses the JNI mechanism described in Section 6.1 to call the C functions.

Figure 7.2 shows a snippet of code that illustrates cross-dependencies between Java and C. The Java snippet contains the *Win32RegKey* class that can retrieve or change values in the Windows registry. The *getValue* method is declared as native, which means that the method is defined outside of the Java language. Also, the *Win32RegKey* class contains two private members: *root* of type *int* and *path* of type *String*. The C snippet contains the implementation of the *getValue* method from the Java side. The prototype of the C function that

implements the Java method conforms to the JNI conventions described in Section 6.1. This example illustrates the use of JNI mechanism for both directions: Java to C, as well as C to Java. The *Java_Win32RegKey_getValue* function gets the definitions from the *Win32RegKey* class using the *GetObjectClass* JNI API function. Then, both private member values of the *Win32RegKey* class are retrieved using the JNI API functions *GetFieldID* and *GetIntField*/*GetObjectField*.

Figure 7.3 contains a snippet of the Java factbase for the *Win32RegKey* application. Some of the fact attributes, such as *line* or *column*, are omitted to focus more on the facts and dependencies used by the algorithms from Figure 6.2 and Figure 6.4.

Figure 7.4 contains a snippet of the C factbase for the *Win32RegKey* application. Some of the fact attributes, such as *line* or *column*, are omitted to focus more on the facts and dependencies used by the algorithms from Figure 6.2 and Figure 6.4.

Figure 7.5 shows a snippet from the cross-dependencies factbase result for the *Win32RegKey* application. Notice that the ids of the nodes in the cross-dependencies factbase are taken from the individual language factbases (in this case from Java and C factbases).

The snippet shows one cross-dependency from a Java method to a C function found by the algorithm from Figure 6.2, and two cross-dependencies from a C function to two Java fields found by the algorithm from Figure 6.4.

The cross-dependency from Java to C illustrated in Figure 7.5 is from the Java method with ID *n6* to the C function with ID *n4*. The C function node has the ID *n4*, the name *Java_Win32RegKey_getValue*, the type *Function*, and the filename *Win32RegKey.c*. This function retrieves for a given key the associated value from the Windows registry. The Java method node has the ID *n6*, the name *getValue*, the type *Method*, its class *Win32RegKey*, and filename *Win32RegKey.java*. The attribute *kind* is *public native*, which means that this method has its definition outside of the Java language. The type of the cross-dependency is *calls*, because it involves a Java method and a C function. The cross-dependency can be read as: Java method "*calls*" the C function. The *mechanism* attribute provides the nature of the cross-dependency (in this case JNI). The *fbfrom* and *fbto* attributes give the factbases from where the ids for the facts involved in the cross-dependency reside. For example, in this cross-dependency, the id *n6* resides in the Java factbase, while the id *n4* resides in the C/C++ factbase.

The two cross-dependencies from C to Java illustrated in Figure 7.5 are from the C function with the ID *n4* (presented also in the cross-dependency from Java to C) to two Java fields. One Java field has ID *n14*, the name *root*, the type *MemberVar*, the class in which is defined *Win32RegKey*, and the filename *Win32RegKey.java*. The other Java field has ID *n15*, the name *path*, the type *MemberVar*, the class in which is defined *Win32RegKey*, and the filename *Win32RegKey.java*. The *root* and *path* Java fields in *Win32RegKey* class indicate the place where the registry values are searched in the Windows

Win32RegKey.java:

```
public class Win32RegKey
{  public Win32RegKey(int theRoot, String thePath)
   {  root = theRoot;
      path = thePath;
   }
   . . .
   public native Object getValue(String name);
   . . .
   private int root;
   private String path;
   . . .
}
```

Win32RegKey.c:

```
JNIEXPORT jobject JNICALL Java_Win32RegKey_getValue
  (JNIEnv* env, jobject this_obj, jstring name)
{  const char* cname;
   jstring path;
   const char* cpath;
   jclass this_class;
   jfieldID id_root;
   jfieldID id_path;
   HKEY root;
   jobject ret;
   char* cret;

   /* get the class */
   this_class = (*env)->GetObjectClass(env, this_obj);

   /* get the field IDs */
   id_root = (*env)->GetFieldID(env, this_class, "root", "I");
   id_path = (*env)->GetFieldID(env, this_class, "path",
      "Ljava/lang/String;");

   /* get the fields */
   root = (HKEY)(*env)->GetIntField(env, this_obj, id_root);
   path = (jstring)(*env)->GetObjectField(env, this_obj,
      id_path);
   . . .
}
```

Figure 7.2: *Win32RegKey* source code snippet

Win32RegKey.java.fb:

```
<?xml version="1.0" encoding="ASCII" ?>
<java xmlns="http://www.javafactbase.org">
 <File id="n1" language="java" name="Win32RegKey.java" />
 <Class id="n3" kind="public" sourcefile="Win32RegKey.java"
  name="Win32RegKey" />
 <Method id="n4" argnames="theRoot,thePath" argtypes="int,String"
  kind="constructor public" inclass="Win32RegKey" sourcefile
  ="Win32RegKey.java" name="Win32RegKey" />
 <Method id="n6" argnames="name" argtypes="String" kind
  ="public native" inclass="Win32RegKey" sourcefile="Win32RegKey.java"
  name="getValue" returntype="Object" />
 <MemberVar id="n14" kind="private" inclass="Win32RegKey"
  sourcefile="Win32RegKey.java" name="root" />
 <MemberVar id="n15" kind="private" inclass="Win32RegKey"
  sourcefile="Win32RegKey.java" name="path" />
 . . .
</java>
```

Figure 7.3: Snippets of *Win32RegKey* Java factbase

Win32RegKey.cpp.fb:

```
<?xml version="1.0" encoding="ASCII" ?>
<cpp xmlns="http://www.cppfactbase.org">
 <File id="n1" language="c++" name="Win32RegKey.c" />
 <Function id="n4" argnames="env,this_obj,name" argtypes
  ="JNIEnv *,jobject,jstring" sourcefile="Win32RegKey.c" name
  ="Java_Win32RegKey_getValue" returntype="jobject" />
 <FunctionDecl id="n19" argtypes="JNIEnv *,jclass,const char *,
  const char *" name="GetFieldID" />
 <calls from="n4" to="n19" calledargs="env,this_class,&amp;quot;root&amp;quot;,
  &amp;quot;I&amp;quot;" sourcefile="Win32RegKey.c" line="31" />
 <calls from="n4" to="n19" calledargs="env,this_class,&amp;quot;path&amp;quot;,
  &amp;quot;Ljava/lang/String;&amp;quot;" sourcefile="Win32RegKey.c" line="32" />
 <FunctionDecl id="n20" argtypes="JNIEnv *,jobject,jfieldID"
  name="GetIntField" />
 <calls from="n4" to="n20" calledargs="env,this_obj,id_root" sourcefile
  ="Win32RegKey.c" line="36" />
 <FunctionDecl id="n21" argtypes="JNIEnv *,jobject" name="GetObjectClass"
  />
 <calls from="n4" to="n21" calledargs="env,this_obj"
  sourcefile="Win32RegKey.c" line="28" />
 <FunctionDecl id="n33" argtypes="JNIEnv *,jobject,jfieldID"
  name="GetObjectField" />
 <calls from="n4" to="n33" calledargs="env,this_obj,id_path"
  sourcefile="Win32RegKey.c" line="37" />
 . . .
</cpp>
```

Figure 7.4: Snippets of *Win32RegKey* C factbase

Win32RegKey.crossdeps.fb:

```
<?xml version="1.0" encoding="ASCII" ?>
<crossdeps xmlns="http://www.crossdepsfactbase.org">
 <calls from="n6" fbfrom="java" to="n4" fbto="cpp" mechanism="JNI" />
 <uses from="n4" fbfrom="cpp" to="n14" fbto="java" mechanism="JNI"
  sourcefile="Win32RegKey.c" line="36" />
 <uses from="n4" fbfrom="cpp" to="n15" fbto="java" mechanism="JNI"
  sourcefile="Win32RegKey.c" line= "37" />
 . . .
</crossdeps>
```

Figure 7.5: Snippets of *Win32RegKey* result cross-language dependencies factbase

registry. The type of the two cross-dependencies is *uses*, because each cross-dependency involves a C function and a Java field. The cross-dependency can be read as: C function "*uses*" the Java field.

7.3 Java GNome Test

The goal of this test is to check the scalability and the accuracy of our approach for discovering cross-dependencies between Java and C/C++ languages. Java-GNOME[1] contains a set of Java bindings for accessing GTK and GNOME C libraries from Java applications [JavaGNome]. Developers can then build GNOME applications [GNOME] using the Java programming language. This system involves 1,132 Java source files containing about 103 KLOC and 329 C source files containing 69 KLOC. In this test, we did not consider the Java-GNOME sample Java files that show how to use the framework.

Our approach revealed for Java-GNOME 3,582 connections from Java to C (Java native methods implemented in C) and 22 connections from C to Java (C functions call/use Java methods/member variables). The algorithm for finding Java to C dependencies also notes those Java native methods for which it cannot find an implementation on the C side, as well as those C functions that call/use Java method/member variable not presented on the Java side.

We found 109 wrong connections from a Java method to a C function, and 6 wrong invocations from a C function to a Java method. After analyzing each occurrence manually, we discovered the following mistakes in the Java-GNOME system:

- Developers did not always follow the JNI type conventions when writing the C function implementation of a Java native method. For example, instead of using *jint* they used *int*. This is a lucky case, because coincidentally the two types both represent 32 bits. There were

[1]The version of Java-GNOME used is 8.3.2.

other cases in which the developers forgot a parameter for a C function, or did not follow the JNI conventions for the name of a C function. For example, the Java native method *gdk_point_free(int)* from Java class *org.gnu.gdk.Point* has the incorrect correspondence in C being *Java_org_gnu_gdk_Point_gdk_point_free(JNIEnv *env, jclass cls, jint obj)*.

- Developers sometimes omitted the second parameter of the C function implementation of a Java native method. This will not cause a compilation error, but when the Java native method is invoked, an unsatisfied link error will appear at runtime. For example, the Java native method *setAnchor(int, int)* from class *org.gnu.gnome.CanvasText* has the correspondence in C being *Java_org_gnu_gnome_CanvasText_setAnchor (JNIEnv *env, jint cptr, jint anchor)*, which is missing the JNI object parameter.

- Some Java native methods were missing C implementations. The system will behave in the same way as in the previous case.

- Some C functions invoke Java methods that do not exist on the Java side. This will result in runtime errors when the C function will be called.

The algorithm found all the cross-dependencies from Java to C, and it missed 5 cross-dependencies from C to Java. In these cases the developers retrieved the method ids based on a Java method name and a wrong Java class parameter.

Figure 7.6 illustrates a snapshot of the cross-language dependencies pairs for JavaGNome. The Java nodes are in orange rectangles, while the C nodes are in yellow rectangles. There are two columns representing the source and the destination nodes of the cross-language dependencies. Note that there are some nodes that appear in the left side and do not have a correspondence in the right side. These cases represent some of the errors in JavaGNome. Using this picture a developer can easily fix the errors.

7.4 Standard Perl Modules Test

The goal of this evaluation is to help assess the accuracy of the discovered cross-language dependencies from Perl subroutines to C functions. We evaluated our approach using all the modules provided with the Perl distribution source code. These modules reside in the *ext* subdirectory, and they contain both C and Perl source code. The C code is used to add more functionality to the Perl side. The new C functions are added to the Perl symbol hashtable for the appropriate package, using the external subroutine mechanism described in Section 6.2.

We can compute the Perl external subroutines in another way. Perl internally keeps some attributes about subroutines. Of particular interest is an

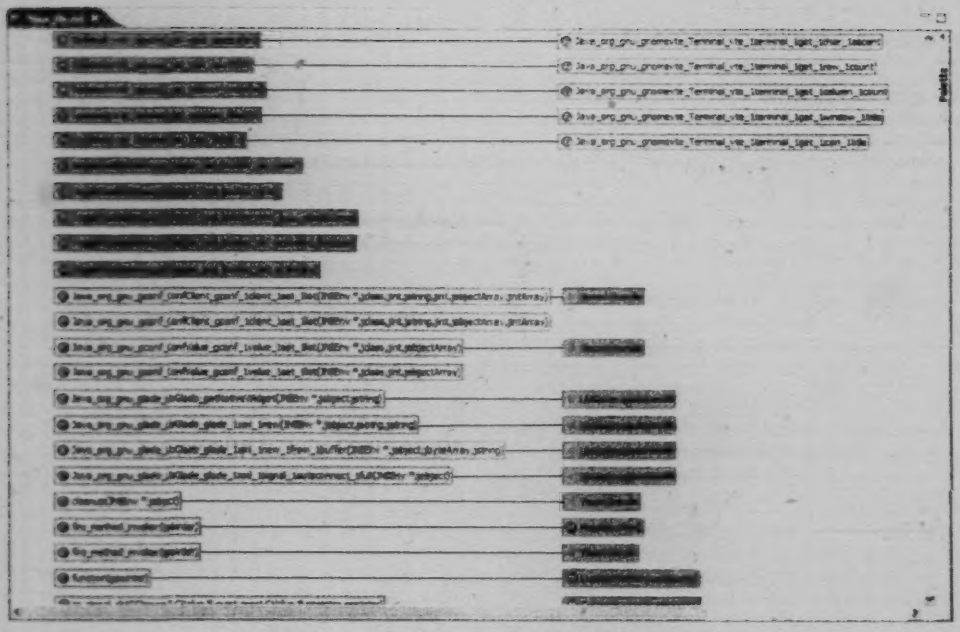

Figure 7.6: Java-GNome mistakes

attribute that, for external subroutines, maintains the address of the C function to be called. For non-external subroutines, this attribute is NULL. The name of the C function can come from debugger information when running the interpreter. Consequently, we know there are at least 853 expected external subroutines (Perl routines paired with C functions).

Our method extracted 851 external subroutines, of which 843 are found also in the alternative way. Thus, there are 10 false negatives (recall 98.8%) and 8 false positives (precision 99.1%). There are a few false positives because the C parser extracts facts before preprocessing; code inside *#ifdef*s is considered, even though it might actually be unused.

7.5 Summary

This chapter presented for tests for evaluation and application of our approach.

The first two tests use small applications, just to illustrate the discovery of the cross-dependencies and the use of the visualization tool. The third test shows how the approach scales on a bigger application, called *Java-GNome*. Our approach proved to be useful to find some of the developers' mistakes.

The last test involves the discovery of cross-dependencies from Perl to C. We evaluated our approach for all the modules provided with the Perl distribution. Some of the Perl subroutines are implemented externally in C language.

Chapter 8

Related Work

This chapter presents different techniques to parse, analyze, and query/visualize the source code for different types of software applications.

Multi-language tool (MT)

Linos et al. [Linos03] implemented a prototype tool, called MT (*Multi-Language Tool*), for understanding multi-language program dependencies. The purpose of MT is to facilitate the process of detecting, storing and managing cross-dependencies found in programs written using a combination of three widespread programming languages: C, C++ and Java. Linos et al. describes via informative examples the interface protocols between the three target languages. The host-to-foreign language dependencies can be identified through specific keywords that the interface protocols implement. MT takes as input several source files, written as a combination of C, C++ and Java, and invokes a parser, which is able to detect the host-to-foreign interfacing information, such as host-to-foreign dependencies, their positions in the source code, number of lines of code or types of files.

MT user interface displays each used programming language as a circle, where the size of the circle is proportional to the corresponding number of lines of code, and the color of the circle indicates whether it is a procedural or an object-oriented language. Inside the circles, various metrics are displayed, such as total number of files or percent of lines of code. Any other programming language except the three languages above-mentioned is displayed using an *Other* circle. The host-to-foreign program dependencies are displayed and the user can edit the source code within MT.

The extractor used in this tool is based on a lexical analysis. Our approach uses a syntactic extractor, which we believe that is more precise for analyzing such heterogeneous systems.

Analysis of distributed multi-language software code

Deruelle et al. [Deruelle01] presented a method for analysis of distributed multi-language software systems. Several tools help to accomplish this: a multi-language source code analyzer, a software change management module,

a profiling tool, and a graphical user interface. The multi-language source code analyzer consists of a set of parsers for each of the languages considered (C, C++ and Java). Each parser is generated using the *JavaCC (Java Compiler Compiler)* tool based on a language specific grammar. The source code could also be byte-code, in which case a decompiler is run first.

The parsers produce an XML-based representation of the source code, called SCSM (Source Code Structural Model), which represents the software components and their relationships. SCSM is able to represent software applications composed of Java or C/C++ source code, relational or object-oriented databases, and CORBA's IDL files and their components. The extracted components and their relationships are stored in a database, called Software Component Repository. The SCMM (Software Change Management Module) propagates a change performed on a component to the others. It is equipped with a facts builder module that inserts facts into a knowledge base, where the facts are the components and their relationships extracted from the structural model SCSM. The propagation is based on change propagation rules that are triggered by inserting facts.

This approach focuses on the CORBA's IDL files, while our approach focuses on the JNI mechanism.

Analyzing COM/COM+ software applications

Pinzger et al. [Pinzger03] presents an approach for analyzing and understanding the COM/COM+ component-based systems. The approach is validated using the Island Hopper application (a three-tired application of about 10 KLOC in which the business and data access logic tiers are COM+ components). For analyzing the COM+ components, the authors used the following information sources: source code for definition (IDL) and source files that define and implement the interfaces of COM+ components, type libraries containing detailed information regarding the interfaces implemented and provided by COM+ components, and Windows Registry that comprises additional information for the configuration of components behavior at deployment time such as transaction semantics and security settings.

The approach starts with the identification of all COM+ components used in the client application by applying the lexical analysis tool called Revealer (based on the method of instantiating the COM+ component that is dependent of the programming language). Having the program identifiers, the authors developed the Component Inspector tool for finding the meta-data of COM+ components (description of external visible interfaces and the corresponding methods) existing in type libraries, and retrieves information about the COM+ components available only at deployment time (such as transaction or security information).

The next step of the process is the source code analysis of the COM+ components using Imagix4D and SourceNavigator tools. The data collecting during the previous steps are stored in Vienna Component Framework data

model, which provides an abstract model to administrate (store and interrogate) the component features (methods, properties, events and so on) in a tree-like data structure. This semi-automatic approach allows maintainers to understand faster the components and their interactions.

Similar to the previous approach, this approach focuses on parsing the IDL files for discovering the COM+ dependencies.

Analyzing Web applications

Hassan et al. [Hassan03] proposed a methodology for helping developers in maintaining their Web applications. A set of parsers is used to analyze the source code of a Web application. The outcome of this analysis is a set of relationships between various components of the Web application. The relationships are abstracted into a box-and-arrow diagram that shows the architecture of the application. The developer gains a better understanding of the system through the visualization of the system architecture. In addition, the developer can interactively explore this diagram, by zooming into the components of interest and querying relationships between participating components.

The components of a Web application considered in this approach were the following: static pages (HTML and JavaScript code), active pages (Active Server Pages), Web objects (DCOM objects) and SQL queries. A set of extractors is used to extract facts by parsing all the source files of the Web application. Five types of extractors have been used: an HTML extractor, a Server Script extractor, a DB Access extractor, a Source Code extractor, and a Binary Code extractor. Based on the type of file, the suitable extractor is used, and the artifacts for every involved language are extracted. A repository called *THEFACTS* stores all the facts from the entire application, as well as the relationships among them. The facts from *THEFACTS* are further abstracted through several levels of abstraction up to the architecture level and their relationships are displayed through a box-and-arrows diagram. The developers can further explore this diagram, seeing the details inside every box at one next lower level of abstraction.

This approach is very similar with our approach in the way that they are using static analysis for discovering the facts from HTML and JavaScript languages.

Analyzing Java applets

Korn et al. [Korn99] built Chava, a tool that extracts artifacts about classes, methods, fields and their relationships, from Java bytecode (in particular Java applets). A bunch of tools for querying, visualizing and analyzing the structural information extracted by Chava were generated automatically from CIAO reverse engineering tool. The structural information extracted by Chava conforms to a data model for Java based on the entity-relationship model that is described in the paper in details. The data model covers all the Java constructs up to the class member level of granularity, so that detailed analysis

using statements or expressions cannot be achieved. Instead, visualization queries such as all relationships or all members of a class could be accomplished using this data model. Also, this approach allows performing forward and backward reachability analysis, as well as showing the differences between two versions of a Java program. The authors described some applications of Chava applied to software systems or even web sites: producing the cluster diagram of a Proxy Server, detecting the potential security flaws (finds all methods that invoke a method whose parent package is java.net), extracting the interactions between Java components, computing different metrics for object oriented design (such as weighted methods per class, depth of inheritance tree, number of children or coupling between object classes), or analyzing website (using Chava to extract the information about every Java applet from a webpage, and the WebCiao system that analyzes a HTML page). An interesting feature of Chava is that of generating the information even from the compiled Java classes.

This approach extracts the facts from Java bytecode, while we extract the facts based on the source files.

Grammar stealing

Around the year 2000, a lot of researchers studied how to develop an automatic tool to replace an old format for the year (only two digits) with a more reliable one formed of 4 digits. The problem was a really serious one, because many financial applications have been using the year field in a lot of computations, which, in the old format, would have produced errors. The errors were caused by the fact that in the old format the year 2000 (represented as 00) was the same as the year 1900 (represented in the same way 00). In the beginning, the problem seemed not too hard to solve, but given the large variety of programming languages, this problem became a complicated one.

Lämmel and Verhoef studied this problem and described the bottlenecks posed by this problem, as well as a viable solution to solve this problem [Lammel01]. It has been estimated that at least 500 programming languages and dialects were available in the public domain, as well as about 200 proprietary languages that private companies have been developing for their own purposes. In order to analyze source code, parsers need to be constructed, and this is a highly time- and resource- consuming process. The 500-Language Problem (500LP problem) refers to the tremendous effort of producing parsers for the existing languages.

Several solutions have been proposed. One idea is to convert from uncommon languages to frequently used languages, but in order to perform this task, a parser is needed. Another solution could be to extract the grammars from the source-code only, but there is no cost-effective method for performing this task. Yet another solution would be to use the compiler parser's output. However, this output does not preserve the original code, since it removes comments, removes macros, etc.

The key to cracking the 500LP problem is to have a cost-effective way (cheap, fast, reliable) of building the grammar for any of the 500+ languages. Once the grammar is built, the parser can be obtained with the help of a parser generation tool.

The solution proposed by Lämmel and Verhoef, called "grammar stealing", refers to the process of recovering the grammar of any language from either the compiler sources or the language reference manual. This procedure is comprised of several steps: automated grammar extraction, sophisticated parsing, automated testing, and automated grammar transformation. Two examples are presented to illustrate each of the steps. According to the authors, this is a much more efficient solution for building a parser for a language both in terms of time and financial expenses.

Agile parsing

Traditionally, software analysis and transformation tools are based on a single grammar. The input source code is parsed, then one or more transformations are applied so that all transformations must conform to the global grammar, and finally the output is generated.

Dean et al. introduced the *agile parsing* paradigm for program software comprehension systems [Dean03]. The agile parsing technique changes the paradigm of parsing based on a single grammar. Agile parsing proposes the use of a customized grammar of the input language that is tailored to each particular analysis and transformation task. The customization of the base grammar is done by overriding the non-terminal definitions according to the task at hand. The input source code is run through a pipeline of transformations, where each transformation has its own customized grammar obtained by overriding the base grammar.

Dean et al. presented the basic techniques of agile parsing in TXL [Dean03]. TXL is a functional programming language for software analysis and transformation. TXL supports agile parsing by supporting two important overriding constructs. The first one is the "redefine" statement, which gives the effective grammar for the tool as the original base grammar with the definition of the overridden non-terminal replaced by the given redefinition. The second one is the "..." notation within a "redefine" statement, which signifies an extension of the original non-terminal definition with other definitions. The paper explains in detail several examples of cases when agile techniques have been applied for software analysis and transformations. For example, the authors describe the automatic translation between two languages, C and Pascal. To conform to TXL, where the input and output of the transformation rules must comply to the grammar, a common (union) grammar for C and Pascal has to be defined to accept both languages. The two grammars are combined at different levels (global declaration level, procedure level, statement level, etc.) using agile parsing.

The key idea of the *agile parsing* paradigm is that the ad-hoc and tool-

based tuning of the base grammar is a more effective and efficient solution for software analysis and transformation than the single-grammar approach. The idea is not restricted to TXL, and could potentially be extended to other transformation languages that have support for customization. However, this solution is tightly dependent on the required task, so that the union grammar and transformation rules have to be tweaked for every particular problem.

Island parsing

Synytskyy et al. acknowledge the problem that parsing is far from being straightforward, when the goal is to parse source files written in multi-language and which contain errors, as in the web applications case [Synytskyy03]. Their idea is to use island parsing to parse the multiple languages input. Island parsing divides the input in two types of categories: ones that are important, called "islands", and others that are less important called "water". The parser will emphasize more on the island parts.

Synytskyy et al. propose a multi-language parser for ASP pages, which are a mixture of HTML, Visual Basic and Java Script languages. The parser is able to build a single parse tree for a given ASP page, by differentiating between the three languages occurring on the page, and by being able to handle intertwined code (for instance, an "if" statement in Visual Basic split by snippets of HTML). The parser is also robust to possible errors in the HTML code, and it only recovers tables, forms, links and client-side scripts tags. As we have mentioned before, the parser is based on island-grammars, which have been shown to perform well at extracting multiple, dissimilar features from documents that also likely contain errors. Each language is represented by one or several island grammars, so that the facts corresponding to each language are likely to be parsed in the correct way. Any input that does not match any of the grammars is treated as uninteresting.

Representing and querying multi-language systems

Kullbach et al. [Kullbach98] introduced a tool that helps the management of inter-program dependencies for a software application developed in various programming languages, database definitions and job control languages. The proposed approach uses a coarse-grained conceptual model for the individual programming languages, based on which, an integrated conceptual model for the multi-language application is developed. The key observation for this specific application is that the inter-program dependencies are defined by job control procedures, since an application is in fact a set of job control procedures and a number of correlating programs and databases. Unifying the individual conceptual models is accomplished according to two rules: 1) similar concepts are generalized into a super-concept; and 2) concepts from different models that have interconnections, are further connected by conceptual relations. The final conceptual model is stored within a repository. Filling the repository with facts from the source code is not a trivial process, because there are a

large number of source files and the conceptual model needs to be invariant to the order the source files are parsed. The conceptual model can be queried using GReQL (Graph REpository Query Language), which is able to reveal the program inter-dependencies.

This approach provides a conceptual model and a query tool to extract cross-dependencies. However, the paper does not provide real cases for finding different connections among different languages.

XOgastan

Antoniol et al. [Antoniol03] present a tool, called XOgastan, based on a novel capability of the gcc/g++ compiler, i.e., saving to a file the ASCII representation of the AST (abstract syntax tree) for each compiled source file. Xogastan differs from previous tools in that in only uses the gcc/g++ output as it is, without modifying the gcc/g++ compiler source code. The gcc/g++ output is translated into GXL [GXL]. The AST produced by gcc/g++ contains different node types, where each node has an id, attributes, and a list of possible linked nodes. A set of translation rules is used to translate each AST node into a GXL element. XOgastan relies on an object-oriented representation of the AST. A note here is that the XOgastan is able to analyze only C++ code, because only the g++ output can be used (the gcc output is useless for AST analysis).

Relying on the compiler facilities is advantageous because compilers evolve as the programming languages evolve. However, the output of the compiler may not be readily suitable for analysis purposes [Lammel01]. In addition, few compilers dump the AST, and working with the compiler's source code to recover the AST may be difficult.

Chapter 9

Conclusions and Future Work

Nowadays, the abundance of new technologies used to ease the development of medium or large applications raises new challenges for understanding and maintaining these systems. In particular, a large number of software systems are developed in more than one implementation language, ranging from programming languages such as C, C++ or Java, to scripting languages such as Tcl, Perl, or JavaScript. This situation motivates the need to adapt existing fact extractors that have been developed for a single language to multi-language scenarios, where not only artifacts pertaining to a certain language are of interest, but also their interconnections.

This thesis presented an approach for revealing the cross-dependencies among several languages.

The first step of our approach builds extractors for each individual language from the subject system. We developed extractors for the following languages: C/C++, Java and Perl. We chose C/C++ and Java for programming languages to show the tightly-coupled connection among programming languages. We chose Perl to illustrate how to build an extractor for scripting languages, and to show the dependencies between a programming language, such as C, and a scripting language, such as Perl. The first step produces one factbase for each individual language in the heterogeneous system analyzed.

In the second step, we developed an analyzer that recognizes the cross-language dependencies among the factbases obtained in the first step. This step assumes that the factbases obtained in the first step conform to a common schema. Essentially, a multi-language system can be represented as a set of namespaces, with each containing facts from one language. The schema helps to decouple the cross-language dependency analysis (and downstream tools like visualizers) from the individual language fact extractors.

To explore and visualize the cross-language dependencies, we developed the Clare tool. Clare is implemented as a plug-in for Eclipse IDE, and shows the facts from each individual language, as well as the cross-language dependencies among the languages. Clare can be used to navigate through the call tree of a diverse system without having the limitation of stopping at the border between

the languages.

We believe that the approach for extracting facts from Perl code may be used to extract facts from other scripting languages, such as Tcl or Python. Also, the steps used to discover the cross-dependencies from a scripting language to C have commonalities, which can be used to develop a generalized technique.

There are a number of directions to proceed with this work. One is to investigate other kinds of tightly-coupled dependencies, such as those caused by embedding an interpreter into a host application written in C. Another is to evaluate how these cross-language dependencies can be made more useful to programmers. Also, we are interested in developing the needed infrastructure to better integrate our scripting language fact extractors into the Eclipse environment. Currently Eclipse provides both JDT (Java Development Tools) and CDT (C++ Development tools) for conventional programming languages.

We also propose to extend our approach for additional programming languages, expand on the control and data integration mechanisms to be recognized, generalize to using the output of other fact extractors, and consider recognizers that operate on the Clare XML-based format directly.

Bibliography

[Affrus] Late Night Software Ltd., Affrus. `http://www.latenightsw.com/affrus`.

[Antoniol03] Giuliano Antoniol, Massimiliano Di Penta, Gianluca Masone, and Umberto Villano. XOgastan: XML-Oriented gcc AST Analysis and Transformations. In *Third IEEE International Workshop on Source Code Analysis and Manipulation*, pages 173–182. IEEE Computer Society, 2003.

[Baxter] Ira Baxter. Specialized Analysis and Modification Tools. `http://www.semdesigns.com/Products/Tools.html`.

[Bowman99] Ivan T. Bowman, Richard C. Holt, and Neil V. Brewster. Linux as a Case Study: Its Extracted Software Architecture. In *Proceedings of the 21st international conference on Software engineering*, pages 555–563. IEEE Computer Society Press, 1999.

[CLR] Microsoft Common Language Runtime (CLR). `http://msdn.microsoft.com/netframework/programming/clr`.

[CNN] CNN: How Soviets copied America's best bomber during WWII. `http://archives.cnn.com/2001/US/01/25/smithsonian.cold.war`.

[CORBA] OMG CORBA. `http://www.corba.org`.

[CPPX] University of Waterloo, CPPX. `http://www.swag.uwaterloo.ca/~cppx`.

[Carroll88] Paul B. Carroll. Computer glitch: Patching up software occupies programmers and disables systems. *Wall Street Journal*, page 1, 1988.

[Chikofsky90] Elliot J. Chikofsky and James H. Cross II. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*, 7(1):13–17, 1990.

[Columbus] FrontEndArt Software Ltd., Columbus/CAN. `http://www.frontendart.com`.

[Corbi89] T. A. Corbi. Program understanding: challenge for the 1990's. *IBM System Journal*, 28(2):294–306, 1989.

[CoreJava2] Cay S. Horstmann and Gary Cornell. *Core Java 2 - Advanced Features*. Sun Microsystems Press, 2000. Chapter 11 - Native Methods.

[DMM] University of Ottawa, Dagshtuhl Middle Model (DMM). `http://scgwiki.iam.unibe.ch:8080/Exchange/2`.

[Dean01] Thomas R. Dean, Andrew J. Malton, and Richard C. Holt. Union Schemas as a Basis for a C++ Extractor. In *Working Conference on Reverse Engineering*, pages 59–68, 2001.

[Dean03] Thomas R. Dean, James R. Cordy, Andrew J. Malton, and Kevin A. Schneider. Agile Parsing in TXL. *Automated Software Engineering*, 10(4):311–336, 2003.

[Deruelle01] L. Deruelle, N. Melab, M. Bouneffa, and H. Basson. Analysis and manipulation of distributed multi-language software code. In *International Workshop on Source Code Analysis and Manipulation*, pages 43–54, 2001.

[Eclipse] Eclipse. `http://www.eclipse.org`.

[Eichberg05] Michael Eichberg, Michael Haupt, Mira Mezini, and Thorsten Schäfer. Comprehensive Software Understanding with SEXTANT. In *International Conference on Software Maintenance*, pages 315–324, 2005.

[Epic] Epic Eclipse plugin IDE. `http://e-p-i-c.sourceforge.net`.

[Ferenc01] Rudolf Ferenc, Susan Elliott Sim, Richard C. Holt, Rainer Koschke, and Tibor Gyimóthy. Towards a Standard Schema for C/C++. In *Working Conference on Reverse Engineering*, pages 49–58, 2001.

[Ferenc02] Rudolf Ferenc, Arpad Beszédes, Mikko Tarkiainen, and Tibor Gyimóthy. Columbus - Reverse Engineering Tool and Schema for C++. In *International Conference on Software Maintenance*, pages 172–181, 2002.

[GNOME] GNOME. `http://www.gnome.org`.

[GXL] GUPRO, Graph eXchange Language (GXL). `http://www.gupro.de/GXL`.

[Hassan01] Ahmed E. Hassan and Richard C. Holt. Towards a better understanding of Web applications. In *International Workshop on Web Site Evolution*, pages 112–116, 2001.

[Hassan02a] Ahmed E. Hassan and Richard C. Holt. Architecture Recovery of Web Applications. In *International Conference on Software Engineering*, pages 349–359, 2002.

[Hassan02b] Ahmed E. Hassan. Architecture Recovery of Web Applications. Master's thesis, Department of Computer Science, Faculty of Mathematics, University of Waterloo, Ontario, Canada, 2002.

[Hassan03] Ahmed E. Hassan and Richard C. Holt. A Visual Architectural Approach to Maintaining Web Applications. *Annals of Software Engineering – Special Volume on Software Visualization*, 16, 2003.

[Holt00] Richard C. Holt, Ahmed E. Hassan, Bruno Laguë, Sébastien Lapierre, and Charles Leduc. E/R Schema for the Datrix C/C++/Java Exchange Format. In *Working Conference on Reverse Engineering*, pages 284–286, 2000.

[IDL] OMG IDL. `http://www.omg.org/gettingstarted/omg_idl.htm`.

[IEEE83] ANSI/IEEE. IEEE standard glossary of software engineering terminology. *IEEE Standard 729*, page 32, 1983.

[JNI] Java Native Interface (JNI). `http://java.sun.com/docs/books/tutorial/native1.1`.

[JavaGNome] Java-GNOME Bindings. `http://sourceforge.net/projects/java-gnome`.

[Jones98] Capers Jones. *Estimating Software Costs*. McGraw-Hill, New York, 1998.

[Ko05] Andrew J. Ko, Htet Aung, and Brad A. Myers. Eliciting design requirements for maintenance-oriented IDEs: A detailed study of corrective and perfective maintenance tasks. In *ICSE '05: Proceedings of the 27th International Conference on Software Engineering*, pages 126–135, 2005.

[Komodo] Komodo IDE. `http://www.activestate.com/Products/Komodo`.

[Korn99] Jeffrey L. Korn, Yih-Farn Chen, and Eleftherios Koutsofios. Chava: Reverse Engineering and Tracking of Java Applets. In *Working Conference on Reverse Engineering*, pages 314–325, 1999.

[Kullbach98] Bernt Kullbach, Andreas Winter, Peter Dahm, and Jürgen Ebert. Program comprehension in multi-language systems. In *Working Conference on Reverse Engineering*, pages 135–143, 1998.

[Lammel01] Ralf Lämmel and Chris Verhoef. Cracking the 500-Language Problem. *IEEE Softw.*, 18(6):78–88, 2001.

[Linos03] Panagiotis K. Linos, Zhi hong Chen, Seth Berrier, and Brian O'Rourke. A tool for understanding multi-language program dependencies. In *International Workshop on Program Comprehension*, pages 64–72, 2003.

[Moise03] Daniel L. Moise and Kenny Wong. An Industrial Experience in Reverse Engineering. In *Working Conference on Reverse Engineering*, pages 275–284, 2003.

[Moise04a] Daniel L. Moise, Kenny Wong, and Dabo Sun. Integrating a Reverse Engineering Tool with Microsoft Visual Studio .NET. In *European Conference on Software Maintenance and Reengineering*, pages 85–94, 2004.

[Moise04b] Daniel L. Moise and Kenny Wong. Issues in Integrating Schemas for Reverse Engineering. *Electronic Notes in Theoretical Computer Science*, 94, 2004.

[Moise05] Daniel L. Moise and Kenny Wong. Extracting and Representing Cross-Language Dependencies in Diverse Software Systems. In *Working Conference on Reverse Engineering*, pages 209–218, 2005.

[Moise06a] Daniel L. Moise, Kenny Wong, H. James Hoover, and Daqing Hou. Reverse Engineering Scripting Language Extensions. In *International Conference on Program Comprehension*, pages 295–306, 2006.

[Moise06b] Daniel L. Moise and Kenny Wong. Perl Fact Extractor. In *Working Conference on Reverse Engineering*, pages 243–252, 2006.

[Muller93] Hausi A. Müller, Mehmet A. Orgun, Scott R. Tilley, and James S. Uhl. A Reverse Engineering Approach to Subsystem Structure Identification. *Journal of Software Maintenance: Research and Practice*, 5(4):181–204, 1993.

[PERL] Perl scripting language. `http://www.perl.org`.

[PYTHON] Python scripting language. `http://www.python.org`.

[Parikh83] Girish Parikh and Nicholas Zvegintzov. *Tutorial on Software Maintenance*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1983.

[Parikh87] Girish Parikh. Making the Immortal Language Work. *International Computer Programs Business Software Review 7*, (33), April 1987.

[Parrot] Allison Randal and Dan Sugalski and Leopold Tötsch. *Perl 6 and Parrot Essentials*. O'Reilly, 2nd edition, 2004.

[PerlBible] Larry Wall and Tom Christiansen and Jon Orwant. *Programming Perl*. O'Reilly, 3rd edition, 2000.

[Pinzger03] Martin Pinzger, Johann Oberleitner, and Harald Gall. Analyzing and Understanding Architectural Characteristics of COM+ Components. In *Proceedings of the 11th IEEE International Workshop on Program Comprehension*, page 54. IEEE Computer Society, 2003.

[Rigi] University of Victoria, Rigi. `http://www.rigi.csc.uvic.ca`.

[RigiManual] Kenny Wong. *Rigi User's Manual*, 1998.

[Riva00] Claudio Riva. Reverse Architecting: An Industrial Experience Report. In *Working Conference on Reverse Engineering*, pages 42–51, 2000.

[SHRIMP] University of Victoria, SHriMP. `http://www.thechiselgroup.org/shrimp`.

[SN] Source Navigator. http://sourcenav.sourceforge.net.

[SNE] Source Navigator Extensions. http://www.wellcode.com/home/modules.php?name=Content&pa=showpage&pid=4.

[SWIG] Simplified Wrapper and Interface Generator (SWIG). http://www.swig.org.

[Sim02] Susan Elliot Sim, Richard C. Holt, and Steve Easterbrook. On Using a Benchmark to Evaluate C++ Extractors. In *International Workshop on Program Comprehension*, pages 114–123, 2002.

[Singer97] Janice Singer, Timothy Lethbridge, Norman Vinson, and Nicolas Anquetil. An examination of software engineering work practices. In *IBM Center for Advanced Studies Conference*, pages 21–36, 1997.

[Synytskyy03] Nikita Synytskyy, James R. Cordy, and Thomas R. Dean. Robust multilingual parsing using island grammars. In *Proceedings of the 2003 conference of the Centre for Advanced Studies conference on Collaborative research*, pages 266–278. IBM Press, 2003.

[TA] Richard C. Holt. An Introduction to TA: The Tuple-Attribute Language. University of Waterloo, 1997.

[TCL] Tcl scripting language. http://www.tcl.tk.

[Tilley94] Scott R. Tilley, Kenny Wong, Margaret D. Storey, and Hausi A. Müller. Programmable Reverse Engineering. *Journal of Software Engineering and Knowledge Engineering*, 4(4):501–520, 1994.

[Winter02] Andreas Winter, Bernt Kullbach, and Volker Riediger. An Overview of the GXL Graph Exchange Language. In *Software Visualization, LNCS 2269*, pages 324–336, 2002.

[Wong95] Kenny Wong, Scott R. Tilley, Hausi A. Muller, and Margaret-Anne D. Storey. Structural Redocumentation: A Case Study. In *IEEE Software*, pages 46–54, 1995.

[XMI] XML Metadata Interchange (XMI). http://www.omg.org/technology/documents/formal/xmi.htm.

[XML] World Wide Web (W3C), Extensible Markup Language (XML). http://www.w3.org/XML.

[XS] Perl XS. http://perldoc.perl.org/perlxs.html.

[XSD] XML Schema (XSD). http://www.w3.org/XML/Schema.